PenTest magazine

starter kit

80 PAGES

Vol.1 No.4 ISSN 2084-1116
Issue 04/2013(04)

Basic UNIX and LINUX Concepts

INTRODUCTION TO LINUX AND UNIX

SECURING WEB AND DNS

LINUX JOB MANAGEMENT

SIGNALS AND INTERRUPT HANDLERS

PLUS

PRIVACY AND ANONYMITY TECHNIQUES TODAY

ISO27001:2013 WHAT HAS CHANGED?

titania

Cyber Security Auditing Software

Improve your Firewall Auditing

As a penetration tester you have to be an expert in multiple technologies. Typically you are auditing systems installed and maintained by experienced people, often protective of their own methods and technologies. On any particular assessment testers may have to perform an analysis of Windows systems, UNIX systems, web applications, databases, wireless networking and a variety of network protocols and firewall devices. Any security issues identified within those technologies will then have to be explained in a way that both management and system maintainers can understand.

The network scanning phase of a penetration assessment will quickly identify a number of security weaknesses and services running on the scanned systems. This enables a tester to quickly focus on potentially vulnerable systems and services using a variety of tools that are designed to probe and examine them in more detail e.g. web service query tools. However this is only part of the picture and a more thorough analysis of most systems will involve having administrative access in order to examine in detail how they have been configured. In the case of firewalls, switches, routers and other infrastructure devices this could mean manually reviewing the configuration files saved from a wide variety of devices.

Although various tools exist that can examine some elements of a configuration, the assessment would typically end up being a largely manual process. Nipper Studio is a tool that enables penetration testers, and non-security professionals, to quickly perform a detailed analysis of network infrastructure devices. Nipper Studio does this by examining the actual configuration of the device, enabling a much more comprehensive and precise audit than a scanner could ever achieve.

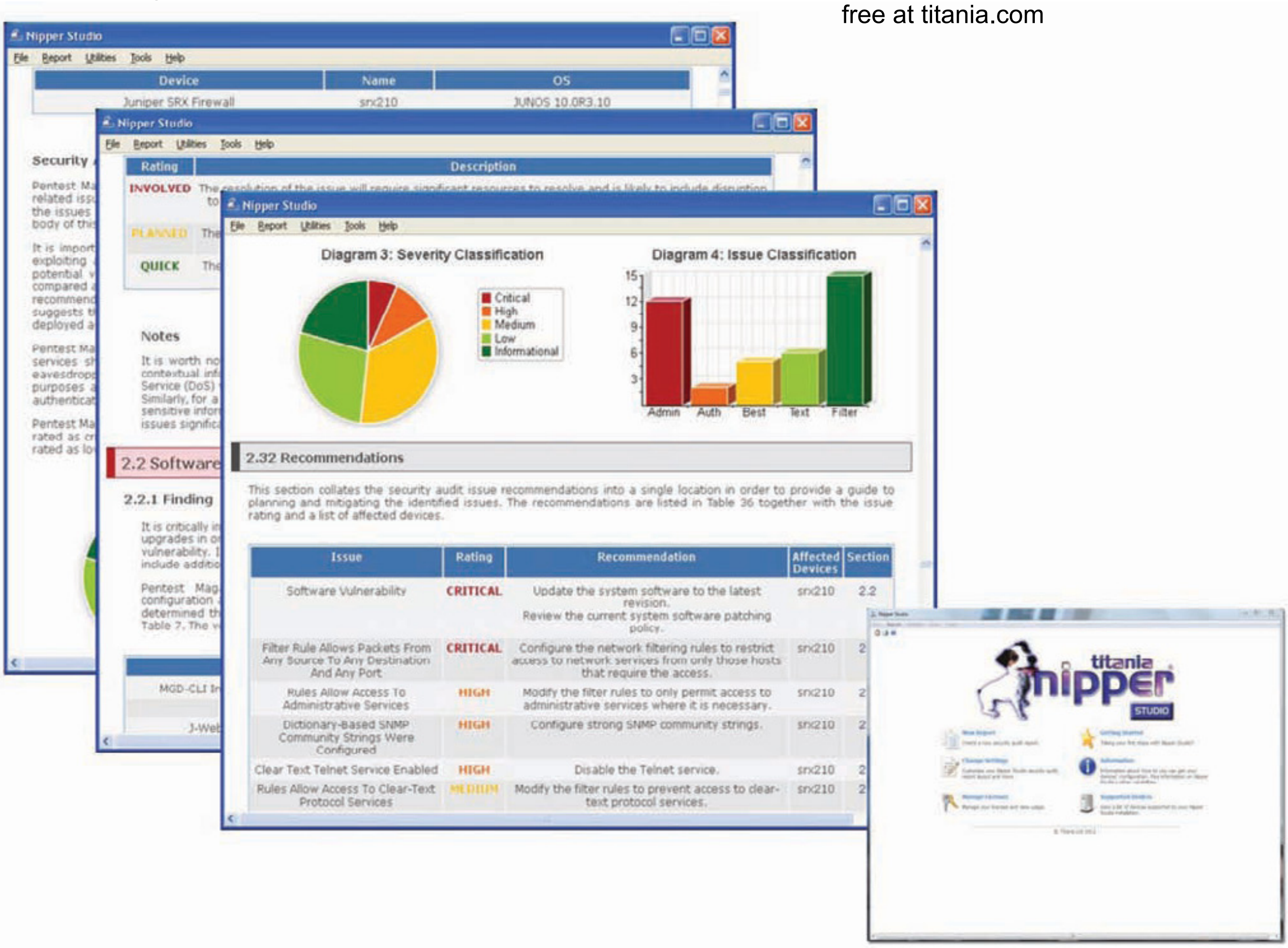
Device Auditing	Scanners	Nipper Studio
Audit without Network Traffic	✗	✓
Authentication Configuration	✗	✓
Authorization Configuration	✗	✓
Accounting/Logging Configuration	✗	✓
Intrusion Detection/Prevention Configuration	✗	✓
Password Encryption Settings	✗	✓
Timeout Configuration	✗	✓
Physical Port Audit	✗	✓
Routing Configuration	✗	✓
VLAN Configuration	✗	✓
Network Address Translation	✗	✓
Network Protocols	✗	✓
Device Specific Options	✗	✓
Time Synchronization	✗	✓
Warning Messages (Banners)	✓*	✓
Network Administration Services	✓*	✓
Network Service Analysis	✓*	✓
Password Strength Assessment	✓*	✓
Software Vulnerability Analysis	✓*	✓
Network Filtering (ACL) Audit	✓*	✓
Wireless Networking	✓*	✓
VPN Configuration	✓*	✓

* Limitations and constraints will prevent a detailed audit

With Nipper Studio penetration testers can be experts in every device that the software supports, giving them the ability to identify device, version and configuration specific issues without having to manually reference multiple sources of information. With support for around 100 firewalls, routers, switches and other infrastructure devices, you can speed up the audit process without compromising the detail.

You can customize the audit policy for your customer's specific requirements (e.g. password policy), audit the device to that policy and then create the report detailing the issues identified. The reports can include device specific mitigation actions and be customized with your own companies styling. Each report can then be saved in a variety of formats for management of the issues.

Why not see for yourself, evaluate for free at titania.com



Ian has been working with leading global organizations and government agencies to help improve computer security for more than a decade.

He has been accredited by CESG for his security and team leading expertise for over 5 years. In 2009 Ian Whiting founded Titania with the aim of producing security auditing software products that can be used by non-security specialists and provide the detailed analysis that traditionally only an experienced penetration tester could achieve. Today Titania's products are used in over 40 countries by government and military agencies, financial institutions, telecommunications companies, national infrastructure organizations and auditing companies, to help them secure critical systems.

Basic UNIX and LINUX Concepts

Table of Contents

Plus

Dear PenTest Readers,

We are proud to present you our 4th PenTest StarterKit “Basic Unix and Linux concepts” which will equip you with valuable knowledge and guide you through various aspects of those commonly used operating systems.This time we will provide you with seventeen articles which will help you to learn a thing or two about Unix/Linux, including our special PLUS. Want to know more? Tune in!

First of all, Nitin Kanoija, Mark Sitkowski and Samanvay Gupta will introduce you to the very bacisc of both operating systems, facilitating your grasping the essence and learning the most important fundamentals. For more details refer to Introduction to “Unix and Linux”, “Introduction to Unix Kernel” and “Unix Basics”.

In the “Securing Web and DNS “ together with our expert Toki Winter you’ll learn several methods for helping to secure those platforms, to find out how to build the software from source and last but not least master your skills on performing an initial secure configuration. After that Alessandro Parisi will prove that Splunk is a must-have tool for gaining a complete and wider picture about what is going on when it cames to preventing security threats.

Advanced users will have a chance to upgrade their skills as well together with Steve Poulsen and his „Linux Job Management”, where you will explore several aspects of the Linux “job,” which will allow the reader to control Linux processes in more time-saving ways.

And last but not least, in our special PLUS section you’re granted with a wonderful and unique possibility to see the intriguing insight into the essance of present security through the eyes of our expert and aWorld Top 10 hacker with 15 years’ experience, Adrian Lamo. Don’t miss his “Privacy and Anonymity Techniques Today”.

All in all, no matter what level of skills, experience and knowledge you possess, we did our best to provide you with vast and various material that will help readers with various background to gain new knowledge and acquire new skills.

Enjoy your reading! Enjoy PenTesting!

Svetlana Balashova and the PenTest Team

Editor in Chief: Ewa Duranc
ewa.duranc@pentestmag.com

Managing Editor: Svetlana Balashova
svetlana.balashova@software.com.pl

Editorial Advisory Board: Jeff Weaver, Rebecca Wynn

DTP: Ireneusz Pogroszewski
Art Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@pentestmag.com

Betatesters & Proofreaders: Michał Rogaczewski, Gilles Lami, John Webb, Kishore P.V, Gregory Chrysanthou, Elia Pinto, Tony Rodrigues, Dallas Moore, Steve Hodge, Amit Chugh, Ivan Gutierrez Agramont, Jarvis Simpson, Joshua Bartolomie

Special Thanks to the Beta testers and Proofreaders who helped us with this issue. Without their assistance there would not be a PenTest magazine.

Senior Consultant/Publisher: Paweł Marciniak

CEO: Ewa Dudzic
ewa.dudzic@software.com.pl

Production Director: Andrzej Kuca
andrzej.kuca@software.com.pl

Art Director: Ireneusz Pogroszewski
ireneusz.pogroszewski@software.com.pl
DTP: Ireneusz Pogroszewski

Publisher: Hakin9 Media
02-682 Warszawa, ul. Bokserska 1
Phone: 1 917 338 3631
www.pentestmag.com

DISCLAIMER!
The techniques described in our articles may only be used in private, local networks. The editors hold no responsibility for misuse of the presented techniques or consequent data loss.

Introduction to Unix Kernel

by Mark Sitkowski

It is usually a source of wonderment to PC users, that the whole of the Unix operating system is in one executable. Instead of a hodge-podge of dll's, drivers, and various occasionally-cooperating executables, everything is done by the Unix kernel.

When Unix was first introduced, the operating system was described as having a 'shell', or user interface, which surrounded a 'kernel' which interpreted the commands passed to it from the shell.

With the passage of time, and the advent of graphical window systems on Apollo and Sun computers, in 1983, this model became a bit strained at the edges. However, it still provides a useful mental image of the system, and window systems can be thought of as a 'candy coat' around the shell. In fact, it isn't just X-windows, which has a direct path to the kernel, since TCP/IP also falls into this category.

The following is not intended to be an exhaustive treatise on the inner workings of the Unix kernel, nor is it specific to any particular brand of Unix. However, it is essential to broadly understand certain important functions of the kernel, before we can take advantage of some of its features, to improve the way in which it handles our process.

The Unix kernel possesses the following functionality, much of which is of interest to us, in pursuit of this goal:

System calls

All of the most basic operating system commands are performed directly by the kernel. These include:

- `open()`
- `close()`
- `dup()`
- `read()`
- `write()`
- `fcntl()`
- `ioctl()`
- `fork()`
- `exec()`
- `kill()`

Since the above commands are actually executed by the kernel, the 'C' compiler doesn't need to generate any actual machine code to perform the function. It merely places a 'hook' in the executable, which instructs the kernel where to find the function.

Modern, third party compilers, however, are ported to a variety of operating systems, and will generate machine code for a dummy function, which itself contains the 'hook'.

Process scheduling and control

The kernel determines which processes will run, when and for how long. We will examine this mechanism, in detail, later.

Networking

That, which became networking, was originally designed so that processes could communicate. It is this ability, to pass information quickly and efficiently from one running process to another, which makes the Unix operating system uniquely capable of multi-dimentional operation. All of the most important communication commands are system calls.

- socket()
- connect()
- bind()
- listen()
- accept()
- send()
- recv()

Device drivers for all supported hardware devices

Unlike other operating systems, where device drivers are separate programs, which are individually loaded into memory, the Unix kernel inherently contains all of the machine's drivers. Contrary to what may be supposed, the entries in the /dev directory are not drivers, they are just access points into the appropriate kernel routine. We will not concern ourselves unduly with the Unix device drivers, as they are outside the scope of this book.

Anatomy of a process

Single-threaded:

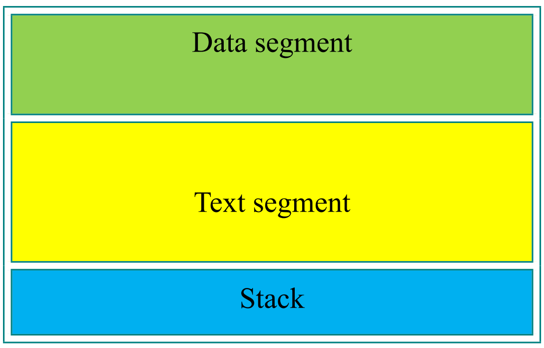


Figure 1. Anatomy of the process – single-threated

Multi-threaded:

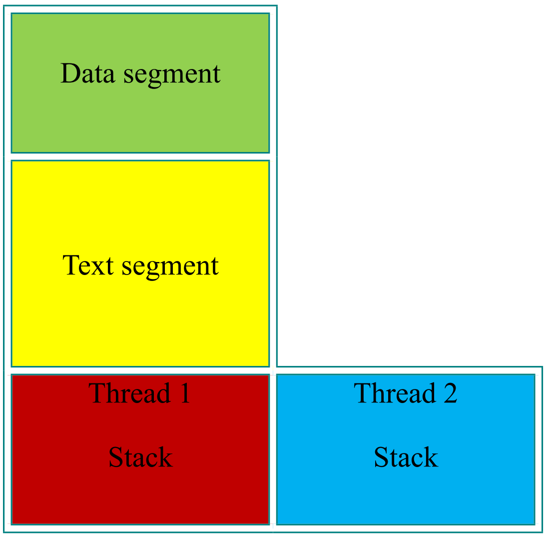


Figure 2. Anatomy of the process – multi-threaded

When an executable is invoked, the following events occur, although not necessarily in this order.

Process Loading

The loader

- Fetches the executable file from the disk
- Allocates memory for all of the global variables and data structures, ('the data segment') and loads the variables into that area of memory.
- Loads the machine code of the executable itself ('the text segment') into memory. With demand-paged executables, this is not strictly the case, as the amount of code actually loaded into memory is several 4k pages. The remainder is put into the swap area of the disk.
- Searches the header portion of the executable, for any dynamically-linked libraries or modules.
- Checks to see if these are already loaded and, If not, the modules are loaded into memory, otherwise, their base addresses are noted.
- Makes available the base addresses of dynamically-linked modules/libraries to the process.
- Allocates an area of memory for the stack. If the process is multi-threaded, a separate stack area is allocated for each thread.

The kernel

- Sets the program counter to the first byte of executable code.
- Allocates a slot in the process table to the new process
- Allocates a process ID to the new process
- Allocates table space for any file descriptors
- Allocates table space for any interrupts.
- Sets the 'ready to run' flag of the process.

All of the above resources, allocated to a given process, constitute the 'context' of a process. Each time the kernel activates a new process, it performs a context switch, by replacing the resources of the previously running process with those of the current one.

At this point, the scheduling algorithm takes over.

The Process Scheduling algorithm

While the process is running, it runs in one of two modes:

Kernel mode

All system calls are executed by the kernel, and not by machine code within the process. Kernel mode has one very desirable characteristic, and that is the fact that system calls are atomic and, hence, cannot be interrupted. One of the most important factors in writing code for high-performance applications, is to ensure that your process executes as much in kernel mode as possible. This way, you can guarantee the maximum CPU time for a given operation.

Kernel threads, such as pthreads, run in kernel mode. It is sometimes worth using a thread, even when doing so doesn't constitute parallel operation, purely to get the advantage of running in kernel mode.

If an interrupt occurs, while in this mode, the kernel will log the signal in the interrupt table, and examine it after the execution of the current system call. Only then will the signal be actioned.

User mode

The ordinary machine code, which makes up much of the executable, runs in user mode. There are no special privileges associated with user mode, and interrupts are handled as they arrive.

It may be seen that, during the time that a process runs, it is constantly switching between kernel mode and user mode. Since the mode switch occurs within the same process context, it is not a computational burden.

A Unix process has one of the following states:

- Sleep
- Run
- Ready to Run
- Terminated

The scheduling algorithm is a finite-state machine, which moves the status of the process between states, depending on certain conditions. Basically, what happens is this.

A process begins to execute. It runs until it needs to perform I/O, then, having initiated the I/O, puts itself to sleep. At this point, the kernel examines the next process table slot and, if the process is ready to run, it enables its execution.

If a process never performs I/O, such as processes which perform long series of floating point calculations, the kernel permits it to only run for a fixed time period, of between 20 and 50 milliseconds, before pre-empting it, and enabling the next eligible process.

When the time comes for a process to run, an additional algorithm determines the priority of one process over another. The system clock sends the kernel an interrupt, once per second, and it is at this time, that the kernel calculates the priorities of each process. Leaving aside the user-level priority weighting, determined by 'nice' the kernel determines priority based on several parameters, of which these are significant:

- How much CPU time the process has previously used
- Whether it is waking up from an I/O wait or not
- Whether it is changing from kernel mode to user mode or not

Swapping and Paging

Of course, the execution of a process is never that straightforward. Only a portion of the code is loaded into memory, meaning that it can only run until another page needs to be fetched from the disk. When this occurs, the process generates a 'page fault' which causes the pager to go and fetch the appropriate page. A similar situation occurs when a branch instruction is executed, which takes the execution point to a page other than those stored in memory. The paging mechanism is fairly intelligent, and contains algorithms similar to those found in CPU machine instruction pipeline controllers. It tries to anticipate branch instructions, and pre-fetch the anticipated page, more or less successfully, depending on the code structure.

Although there are certain similarities, paging, as described above, which is a natural result of process execution, should not be confused with swapping.

If the number of processes grows, to the extent that all available memory becomes used up, the addition of another process will trigger the swapper, and cause it to take a complete process out of memory, and place it in the swap area of the disk.

This is, computationally, an extremely expensive operation, as the entire process, together with its context, has to be written to disk then, when it is permitted once again to run, another process must be swapped out, to make space for it to be reloaded into memory.

So, what does all of this have to do with Performance?

It may be seen from the above, that processes, which are designed for performance-critical applications, should avoid doing physical I/O until it is absolutely necessary, in order to maximize the amount of contiguous CPU time. If it is at all possible, all of the I/O operations should be saved up, until all other processing has completed, and then performed in one operation, preferably, by a separate thread.

As far as threads are concerned, let us consider what happens, when we launch a number of threads, to perform some tasks in parallel.

First, the threads are each allocated a separate stack, but they are not allocated a separate process table slot. This means that, although there are several tasks executing in parallel, this only occurs during the active time of that slot. When the kernel pre-empts the process, execution stops. Multi-threading will not give your application any more system resources.

Further, if we consider a situation, where we have 100 processes running on a machine, and one of them is ours, then we would expect to use 1% of the CPU time. However, if 25 of them are ours, we would be eligible to use 25% of the CPU time.

Thus, If an application can split itself into several processes, running concurrently, then, quite apart from the obvious advantages of parallelism, we will capture more of the machine's resources, just because each child process occupies a separate process table slot.

This also helps, when the kernel assigns priorities to processes. Even though we may be penalized for using a lot of CPU time, the priority of each process is rated against that of other processes. If many of these belong to one application then even though the kernel may decide to give one process priority over another, the application, as a whole, will still get more CPU time.

Additionally, if we are running on a multi-processor machine, then we can almost guarantee to be given a separate CPU for each child process. The kernel may juggle these processes over different CPU's, as a part of its load-balancing operations, but each child will still have its own processor.

The incorporation of the above techniques, into our software architecture forms the cornerstone of multi-dimensional programming.

In Summary

- Each child process gets a CPU different to that used by the parent.
- The more processes contribute to the running of your application, the more CPU time it will get.
- Multi-threading creates multiple execution paths within one process table slot. It may permit parallel execution paths, but it will not get the application more CPU time, or a new CPU.

Therefore:

- Find parallelism within your application. This will make your software run more efficiently, anyway.
- Employ multi-threading where it is not possible to fork a separate process, or where you need to refer to global information, as in the parent process.
- Having decided how the children will communicate the data back to the parent, launch a separate child process for every possible parallel function,to gain the maximum CPU time

System calls

fork()

Under pre-Unix operating systems, starting a process from within another process was traditionally performed as a single operation. One command magically placed the executable into memory, and handed over control and ownership to the operating system, which made the new process run.

Unix doesn’t do that.

Each process has a hierarchical relationship, with its parent, which is the process which brought it to life, and with its child or children which, in turn, are processes which it, itself, started. All such related processes are part of a `process group`.

If a `kill()` signal is sent to the parent of the process group, it will propagate to the child processes.

Unix also has the concept of a ‘session’ which, essentially, can be thought of as comprising all of the process groups associated with a login, or TCP/IP connection.

The basic mechanism, which initiates the birth of a new process is `fork()`.

The `fork()` system call makes a running copy of the process which called it. All memory addresses are re-mapped, and all open file descriptors remain open. Also, file pointers maintain the same file position in the child as they do in the parent.

Consider the following code fragment:

```
pid_t pid;

    switch((pid = fork()){
        case -1:
            printf("fork failed\n");
        break;
        case 0:
            printf("Child process running\n");
```

```
            some_child_function();
        break;
        default:
            printf("Parent process executes this code\n");
        break;
    }
```

At the time that the `fork()` system call is called, there is only one process in existence, that of the expectant parent. The local variable pid is on the stack, probably uninitialised. The system call is executed and, now, there are two identical running processes, both executing the same code. The parent, and the new child process both simultaneously check the variable pid, on the stack. The child finds that the value is zero, and knows, from this, that it is the child. It then executes `some_child_function()`, and continues on a separate execution path.

The parent does not see zero, so it executes the 'default' part of the `switch()` statement. It sees the process ID of the new child process, and drops through the bottom of the `switch()`. Note, that, if we do not call a different function in the case 0: section of the switch, that both parent and child will continue to execute the same code, since the child will also drop through the bottom of the `switch()`.

Programmers who know little about Unix, will have a piece of folklore rattling around in their heads, which says 'a `fork()` is expensive. You have to copy an entire process in memory, which is slow, if the process is large'.

This is true, as far as it goes. There is a memory-to-memory copy of that part of the parent, which is resident in memory, so you may have to wait a few milliseconds.

However, we are not concerned with trivial processes whose total run time is affected by those few milliseconds. We are dealing exclusively with processes whose run times are measured in hours, so we consider a one-time penalty of a few milliseconds to be insignificant.

When a parent forks a child process, on a multi-processor machine, the Unix kernel places the child process onto its own, separate CPU. If the parent forks twelve children, on a twelve CPU machine, each child will run on one of the twelve CPU's.

In an attempt to perform load-balancing, the kernel will shuffle the processes around the CPU's but, basically, they will remain on separate processors.

The `fork()` system call is one of the most useful tools, for the full utilisation of a multi-processor machine's resources, and it should be used whenever one or more functions are called, which can proceed their tasks in parallel. Not only is the total run time reduced to that of the longest-running function, but each function will execute on its own CPU.

vfork()

There is a BSD variant of `fork()`, which was designed to reduce the memory usage overhead associated with copying, possibly, a huge process in memory.

The semantics of `vfork()` are exactly the same as those of `fork()`, but the operation is slightly different. `Vfork()` only copies the page, of the calling process. which is currently in memory but, due to a bug (or feature) permits the two processes to share the same stack. As a result, if the child makes any changes to variables local to the function which called `vfork()`, the changes will be visible to the parent.

Knowledge of this fact has enabled experienced programmers to make use of the advantages of `vfork()`, while avoiding the pitfalls. However, far more subtle bugs also exist, and most Unix vendors recommend that `vfork()` only be used, if it is immediately followed by an `exec()`.

exec()

The original thinking behind `fork()`, was that its primary use would be to create new processes – not just copies of the parent process. The `exec()` system call achieves this, by overlaying the memory image of the

calling process with the new process. There is a very good reason for separating `fork()` and `exec()`, rather than having the equivalent of VMS's `spawn()` function, which combines the two. That reason is, because it is sometimes necessary, or convenient, to perform some operations in between `fork()` and `exec()`. For example, is may be necessary to run the child process as a different user, like root, or to change directory, or both.

There is, in fact, no such call as `exec()`, but there are two main variants, `execl()` and `execv()`.

The semantics of `execl()` are as follows:

```
execl(char *path, char *arg0, char *arg1…char *argn, (char *) NULL)
execv(char *path, char *arg0, char **argv)
```

It may be seen, that the principal difference between the two variants, is that, whereas the `execl()` family takes a path, followed by separate arguments, in a NULL terminated, comma-separated list, the `execv()` variants take a path, and a vector, similar to the `argv[]` vector, passed to a `main()` function.

The first variant of `execl()` and `execv()`, adds an environment vector to the end of the argument list:

```
execle(char *path, char *arg0, .…char *argn, (char *) NULL, char **envp)
execve(char *path, char *arg0, char **argv, char **envp)
```

The second variant replaces the 'path' argument, with a 'file' argument. If this latter contains a slash, it is used as a path, otherwise, the PATH environment variable of the calling process is used to find the file.

```
execlp(char *file, char *arg0, .…char *argn, (char *) NULL, char **envp)
execvp(char *file, char *arg0, char **argv, char **envp)
```

We can now combine `fork()` and `exec()`, to execute lpr from the parent process, In order to print a file:

```
    pid_t pid;

    switch((pid = fork()){
        case -1:
            printf("fork failed\n");
        break;
        case 0:
            printf("Child process running\n");
            execl("/usr/ucb/lpr", "lpr", "/tmp/file", (char *) NULL);
        break;
        default:
            printf("Parent process has executed lpr to print a file\n");
        break;
    }
```

The above code only has one problem. If the parent process quits, the child process will become an orphan, and be adopted by the 'init' process. When lpr has run to completion, it will become a zombie process, and waste a slot in the process table. The same happens if the child prematurely exits, due to some fault.

There are two solutions to this problem:

We execute one of the `wait()` family of system calls.

A waited-for child does not become a zombie, but the parent must suspend processing, until the child terminates, which may or may not be a disadvantage. There are options, which allow processing to continue, during the wait, but the parent needs to poll `waitpid()`, which makes our second solution, described below, a much better option.

If we are waiting for a specific process, the most convenient call is to `waitpid()`. The synopsis of this call is:

```
pid_t waitpid(pid_t pid, int *status, int options);
```

The call to `waitpid()` returns the process ID of the child for which we are waiting, whose process ID is passed in as the first argument, 'pid'.

The second argument, 'status' is the returned child process exit status, and 'options' is the bitwise-OR of the following flags:

WNOHANG: prevents `waitpid()` from causing the parent process to hang, if there is no immediate return.

WNOWAIT: keeps the process, whose status is returned, in a waitable state, so that it may be waited for again.

The options flags are of no use to us, so we set them to zero. The status word, however, provides useful information on how our child terminated, and can be decoded with the macros, described in the man page for 'wstat'.

```
pid_t pid;
int status;

    switch((pid = fork()){
        case -1:
            printf("fork failed\n");
        break;
        case 0:
            printf("Child process running\n");
            execl("/usr/ucb/lpr", "lpr", "/tmp/file", (char *) NULL);
        break;
        default:
            printf("Parent process has executed lpr to print a file\n");
 if(waitpid(pid, &status, 0) == pid){
      printf("lpr has now finished\n");
 }
        break;
    }
```

If we don't wish to poll `waitpid()` repeatedly, but need to do other processing, while the child process goes about its business, then we need to, effectively, disown the child process.

As soon as the child has successfully forked, we must disassociate it from the process group of the parent.

Process groups and sessions are discussed at the beginning of the `fork()` section but, to save you the trouble of looking, a process group is headed by the parent process, whose process ID becomes the group's process group ID. All children of the parent then share this process group ID.

The disowning of a child process is accomplished by executing the system call `setpgrp()`, or `setsid()`, (both of which have the same functionality) as soon as the child is forked. These calls create a new process session group, make the child process the session leader, and set the process group ID to the process ID of the child.

The complete code is as below:

```
pid_t pid;

    switch((pid = fork()){
        case -1:
            printf("fork failed\n");
        break;
        case 0:
 if(setpgrp() == -1){
                printf("Can't set pgrp\n");
            }
```

```
printf("Independent child process running\n");
            execl("/usr/ucb/lpr", "lpr", "/tmp/file", (char *) NULL);
        break;
        default:
            printf("Parent process has executed lpr to print a file\n");
        break;
    }
```

open() close() dup() read() write()

These system calls are primarily concerned with files but, since Unix treats almost everything as a file, most of them can be used on any byte-orientated device, including sockets and pipes.

*int open(char *file, int how, int mode)*

`open()` returns a file descriptor to a file, which it opens for reading, writing or both. The 'file' argument is the file name, with or without a path, while 'how' is the bitwise-OR of some of the following flags defined in fcntl.h:

O_RDONLY Read only

O_WRONLY Write only

O_RDWR Read/write

O_TRUNC Truncate on opening

O_CREAT Create if non-existent

The 'mode' argument is optional, and defines the permissions on the file, using the same flags as chmod.

Int close(int fd)

Closes the file, which was originally opened with the file descriptor fd.

Int dup(int fd)

Returns a file descriptor, which is identical to that passed in as an argument, but with a different number. This call seems fairly useless, at first glance but, in fact, it permits some powerful operations, like bi-directional pipes, where we need a pipe descriptor to become a standard input or output. Also, client-server systems, need to listen for incoming connections on a fixed socket descriptor, while handling existing connections on different descriptors.

About the Author

Mark Sitkowski Design Simulation Systems Ltd http://www.designsim.com.au
Consultant to Forticom Security http://www.forticom.com.au

Automation and Scripting

by Sebastian Perez

Unix/Linux knowledge is something that a security analyst or penetration tester must have. In these days, it is more probable to find Unix/Linux servers within a company; furthermore, several tools exist on UNIX/Linux that could help a security administrator to perform different tasks. In this article, I will talk about automation on these environments. To be more precise, the article will be focused on scripting on Linux environments using bash shell. I would like to start with an introduction of what is scripting before I start coding.

What is Scripting?

Scripting is the art of writing programs for a special run-time environment that can be interpreted by the operating system. The idea of scripting is to automate the execution of tasks that could be executed by an operator. For example, instead of typing all the system commands that are required for compiling and installing an application in Linux, a script could be written to execute these tasks, so the user only needs to execute this script. Scripts are considered high level programing languages and are specific for an operating system/application. For example, Visual Basic scripts may run only in Windows operating system, while shell scripts could be executed in most of the Unix/Linux environments (it depends on the shell used and system architecture). The main difference within programming languages is that scripting language is not compiled; it is interpreted by the operating system's shell. This difference makes scripting languages easy to write and execute, and the source code could be read and analyzed, as this is not compiled (obfuscated); however, this hinders the portability, as the operating system where this script will be executed, should have all the required libraries, commands, etc. There are several scripting languages, depending on the host operating system that is being used.

Shell Scripting

Scripting in Unix/Linux is basically commands stored in a text file that are being interpreted and executed by the shell. It is a tool for building applications by using system calls, tools, utilities, and compiled binaries. The Unix/Linux shell is a command interpreter that sits between the user and the operating system, and it is also a powerful programming language. All the UNIX/Linux commands, utilities, and tools are available for invocation by a shell script. It also provides functionalities as loop constructs, conditionals, and also could evaluate if the last executed command ran successfully or not. Depending on the shell utilized, the scripting syntax will be slightly different. Table 1 shows the most known Unix/Linux shell interpreters and a short review of each of them.

Table 1. Most known shell interpreters in Unix/Linux

Shell name	Description
Sh	Bourne shell, was one of the first shells that provided scripting features
Bash	Bourne again shell. Most common shell in Linux. It was developed to replace the Bourne shell
CSH	C shell. The syntax and usage are very similar to the C programming language
KSH	Korn Shell. It provides some differences with Bourne shell, like Job control, command aliasing, and command history
TCSH	TCSH is an enhanced but completely compatible version of the Berkeley UNIX C shell (CSH)

Each Unix/Linux shell has a different syntax, and also could have different commands that may be executed from the script. In order to easily identify the current shell that is being used, type the following command:

```
echo $SHELL
```

The command output will display the path of the current shell in use. For Linux flavours, it's usually `/bin/bash`, which corresponds to the Bourne Again shell. This article will mainly focus on bash scripting, but most of the syntax could be used within ksh and bsh. Csh is more oriented to C programming language, and usually uses a different syntax, so probably bash scripts will not work within this shell.

Unix/linux provides several text editors that could be used to create a shell script. As vi is available in all these distributions, we will focus on this editor. To start a script, just execute the following command, from the shell:

```
vi scriptname.sh
```

Unix/Linux does not require that the files have an extension, as the operating system identified the type of file based on the file's header. However, it is a good practice to have this extension to easily identify the file type we are working on. Once the previous command was executed, an empty file will be displayed.

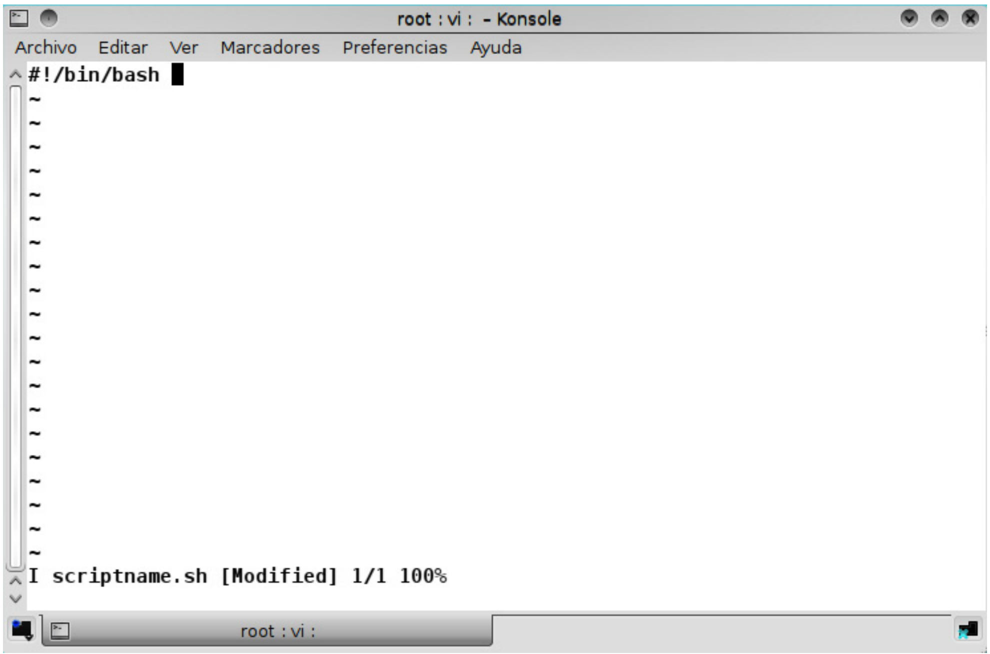


Figure 1. Creating a shell script with vi

The first line that an script should have, is the shell interpreter that will execute this script. In today's shell, this line is not mandatory, but it is a good practice if several scripts for different shell are being developed, to easily identify which is the correct shell to execute this, and avoid compatibility issue due the differences within the syntax. In Figure 1, it can be observed that the shell interpreter used is bash. For those who are not familiar with the vi editor, in order to start writing, just press the key "i" below this first line, where the script starts. Let's start by writing a simple shell that displays who is the user that executes the script, the local time of the host where the script is executed, and the operating system type. The commands gathering this information are *whoami*, *date*, *uname -a*. This article will not explain this commands, as they are basic Unix/Linux commands. The script that executes these instructions is displayed below:

```
#!/bin/bash
whoami
date
uname -a
```

Now, we need to save the script in order to execute it. The proper way to do this, on vi editor, is by pressing the “Esc” key, and the write *wq*. This instructs vi to write the modifications to the file, and quit the editor. Now that the script was saved, it is time to execute this. To do so, first we need to change the script permissions to add the executing permissions. This could be accomplished by executing one of the following commands:

- *chmod 555 scriptname.sh*: it provides read/execute permission to every user
- *chmod +rx scriptname.sh*: it is an alternative way to provide read/execute permission to every user
- *chmod u+rx scriptname.sh*: it provides read/execute permission to the current owner of the script. The owner is the user that created it

The script could also be executed without this permission by executing the bash interpreter and passing the script as a parameter:

```
bash scriptname.sh
```

After executing the script, the commands that are stored within the same are being executed, and the shell will display the command output, as shown in Figure 2. Note that the first line in the script is not executed, or displayed. Every line that starts with an # indicates that this is a comment, and it is not interpreted by the shell.

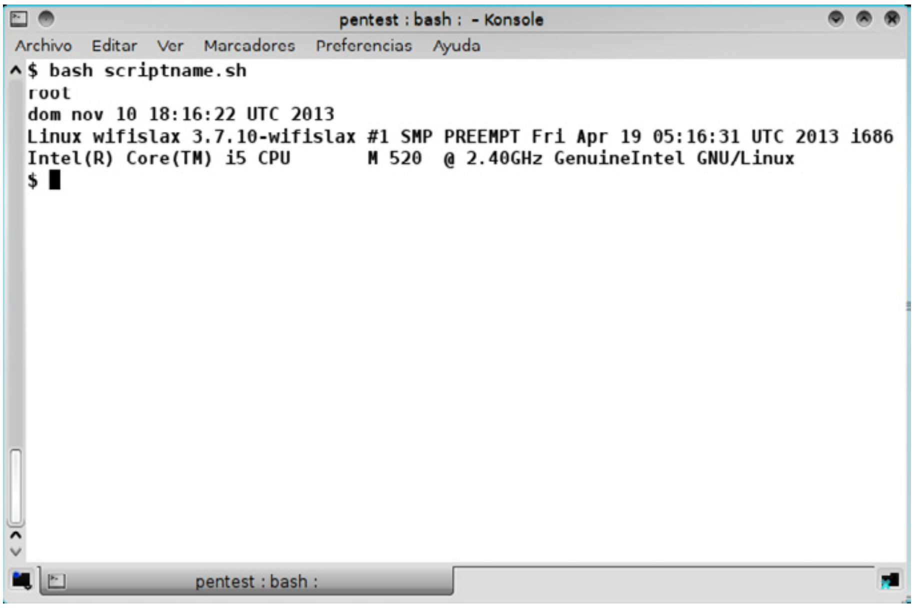


Figure 2. Executing the first script

The first line in the screenshot is the script execution; the second line is the output from the *whoami* command; the third line displays the current date of the system, and the fourth line (which is splitted in two lines) is the details of the current Linux distribution used.

Basic Input/Output Operations

The next thing to learn about scripting is how to communicate with the user. The basic Input/Output commands that a script could interpret are:

- *echo “message”*: displays a message to the user. The message should be specified within the source code of the script. The message could also be a variable, and its content will be displayed
- *read variable*: request the user to provide an input for the script, which will be stored in the defined variable.

The following script will help to understand the basic Input/Output operations that could be performed from an script.

```
#!/bin/bash
echo “User Input required”
read variable1
echo “Content of variable1 is: $variable1”
```

The result from executing this script is shown in Figure 3.

Figure 3. Basic Input/Output operations

Probably it was observed that when the variable1 was invoked within the echo command to be displayed, it had a dollar sign (`$`) at the beginning. This is how the shell identifies that the string that follows this sign is a variable name. If no dollar sign is at the beginning of the variable name, the shell will interpret the string as text, and will display this instead the content of the variable.

The Unix/Linux shells provides ways to communicate different commands within each others, and to perform redirections to/from text files. Every Unix/Linux command has 3 interfaces, standard input (*stdin*), standard output (*stdout*) and standard error (*stderr*).

- Standard input is usually the keyboard, from where the input required for the command to be executed is provided.
- Standard output is usually the screen, where the command displays the results when this was properly executed.
- Standard error is also the screen, where the command displays the errors encountered during its execution

The shell provides a way to redirect these three interfaces to the one required by the user, usually text files. The proper way to use this redirections is with the symbols `< and >`. Below are some examples and explanations of how this redirections are being performed.

- *ls > list.txt*: the *ls* command displays the content of a directory, usually provided as a parameter. If no directory is provided, then it displays the content of the current working directory. The use of the > character instructs the shell to redirect the standard output and store it within the file provided; in this case, *list.txt*. If this is a new file, then the shell will create it; but if the file already existed, it will replace all the previous content in this file, within the command output. If >> is used, this will instruct the shell to append the command output to the file, instead of deleting the content it already had.

- *ls 2> list.txt*: in this case we are using *2>* instead of just >. This indicates the shell to redirect the standard error to the file provided. This allows to create a second file that stores all the errors that were generated during the execution, instead of displaying these on the screen.

- *sort < list.txt*: the sort command will read the standard input provided and will sort it alphabetically. By using the < character, the shell will interpret that the data that must be sorted, the standard input, is the one that is stored within the text file *list.txt*.

Let's suppose that we would need to list the files within the current directory, and then we would like to sort these alphabetically. Using the redirections, the command will be like the following:

- ls > list.txt

- *sort < list.txt*

This will perform the required tasks, however, a new file was created, and displayed within the content of the current directory (Figure 4).

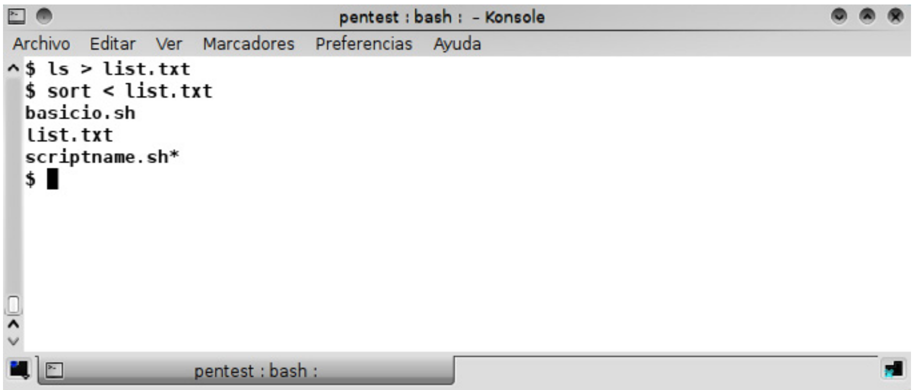


Figure 4. Using input/output redirection

There is another shell feature that could be used in the previous situation, without the need to create a new text file. This feature is called pipe |. This pipe will interconnect two commands by using the standard output from the first command as the standard input for the second command. Considering the previous scenario, using pipes, the command will be like this:

```
ls | sort
```

The pipe works as a temporary file, where the standard output from the *ls* command is stored, and then passed as standard input to the sort command, generating the same result, with the exception of the file created using the standard output redirection (Figure 5).

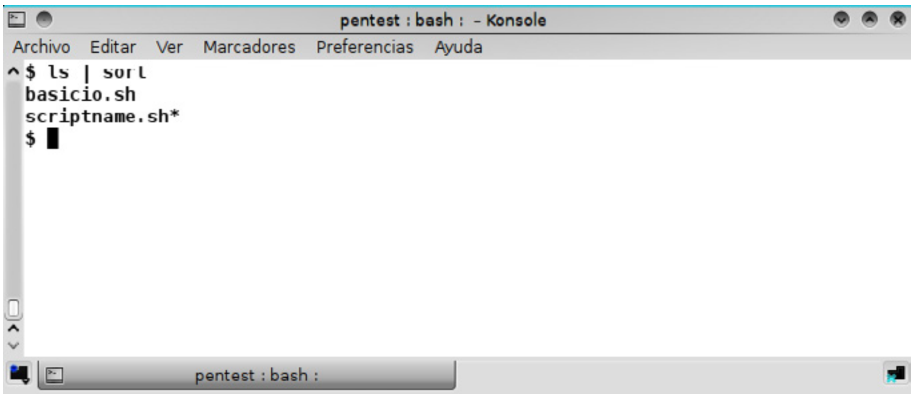


Figure 5. Using pipes to communicate process

Flow Control

The shell also provides, as most of the programming and scripting languages, the capability to control the flow of execution depending on some conditions. These conditions are being evaluated, and depending of the results (true or false), different actions are being performed. The different types of evaluations that could be performed are:

Strings Evaluation

- *string1 = string2*: it is true if both strings are equal
- *string1 == string1*: it is true if both strings are equal
- *string1 != string2*: it is true if both strings are not equal
- *string1 < string2*: it is true if the string1 is less than string2 in ASCII alphabetical order
- *string1 > string2*: it is true if the string1 is greater than string2 in ASCII alphabetical order
- *-z string1*: it is true if string1 is empty
- *string1*: it is true if string1 is not empty
- *-n string1*: it is true if string1 is not empty

Numbers Evaluation

- *number1 -eq number2*: it is true if number1 is equal to number2
- *number1 -ne number2*: it is true if number1 is not equal to number2
- *number1 -gt number2*: it is true if number1 is greater than number2
- *number1 -ge number2*: it is true if number1 is greater than or equal to number2
- *number1 -lt number2*: it is true if number1 is less than number2
- *number1 -le number2*: it is true if number1 is less than or equal to number2

Files Evaluation

- *-e file*: it is true if the file exists
- *-d file*: it is true if the file exists, and it is a directory (for Unix/Linux, everything is a file, including directories, devices, etc.)
- *-f file*: it is true if the file exists, and it is a regular file
- *-L file*: it is true if the file exists and it is a symbolic link (a special type of file that links to an existing file)
- *-r file*: it is true if the file exists and the user who executes the evaluation has read access
- *-w file*: it is true if the file exists and the user who executes the evaluation has write access
- *-x file*: it is true if the file exists and the user who executes the evaluation has execute access

- *file1 -nt file2*: it is true if file1 is newer than file2, according to the modification dates
- *file1 -ot file2*: it is true if file1 is older than file2, according to the modification dates

The condition evaluation could be performed by using *[ ]* or by using the *test* function, for example:

```
test "$string1" = "$string2"
[ "-d file.txt" ]
```

The first flow control we will see is the *IF* statement. This is the most basic flow control; it evaluates a condition, if it is true, it executes the specified commands and if it is not true, it executes the other commands. The use of *else*, and execution *if false* is an optional feature. The syntax is like the following:

```
if condition
then
 execution if true
else
 execution if false
fi
```

Considering a scenario where the user is required to pick a number, and if the number 5 is selected, then a particular message is presented; if not, a different message is shown. The script code for this task will be like the one in Listing 1.

Listing 1. The IF statement

```
#!/bin/bash
echo -n "User Input required:"
read number
if [ $number = 5 ]
then
 echo "You win"
else
 echo "Better luck the next time"
fi
```

Figure 6. Controlling the flow with the IF statement

Figure 6 displays two different execution of the same script. In the first scenario, the condition evaluated by the *IF* was false, and the alternative message was presented. In the second execution, the number picked was 5, so the *IF* evaluation returned true, and the proper message was presented to the user.

The next flow control is the *WHILE* statement. This flow control performs an action while a certain condition is true. Once the condition is false, the *WHILE* statement stops its execution. This statement evaluates the condition before executing the command, so it is possible that this command will never be executed if the condition is never true. The syntax is:

```
while [ condition ]
do
 command1
 command2
 ....
done
```

Let's assume a scenario where the script keeps asking the user to provide a number while the number is less than 10. The script code to perform this will be like the one shown in Listing 2.

Listing 2. WHILE statement

```
#!/bin/bash
echo -n "User Input required:"
read number
while [ $number -lt 10 ]
do
 echo "Please select another number"
 read number
done
```

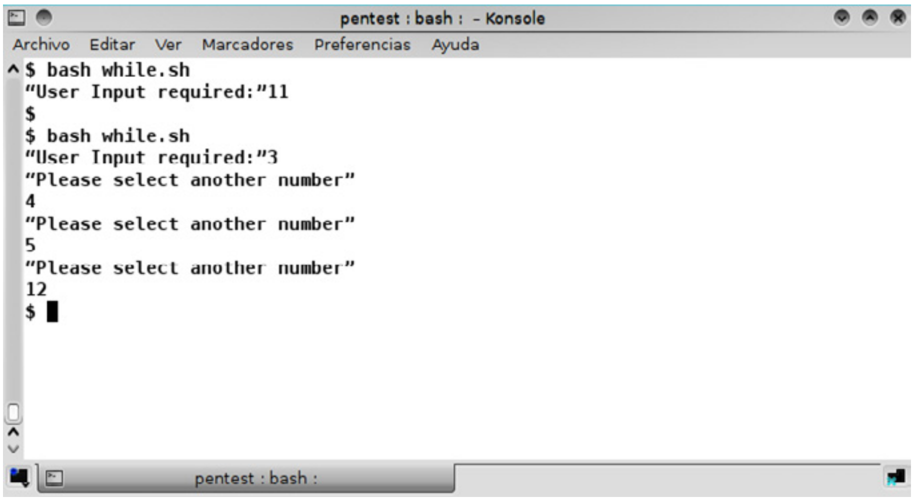


Figure 7. Controlling the flow with the WHILE statement

Figure 7 displays two different executions of the same script. In the first scenario, the condition evaluated was false and the *WHILE* statement was not executed. In the second scenario, the *WHILE* statement was executed three different times.

The next flow control, is the *FOR* statement. This function will execute a determined task a predefined number of times. However, this function does not work as in most programming and scripting languages. In shell scripting, the variable involved in the *FOR* statement will take different values according to the list provided. For example, if we provide a list containing fruits, in every iteration, the variable used will pick a different fruit name, and will stop once all the fruits were picked. The proper syntax is:

```
for variable in list
do
 comand
done
```

Suppose that we need to enumerate the files within the current working directory. The shell script code that performs this action will be:

```
#!/bin/bash
for file in $(ls .)
do
 echo "This is the file $file"
done
```

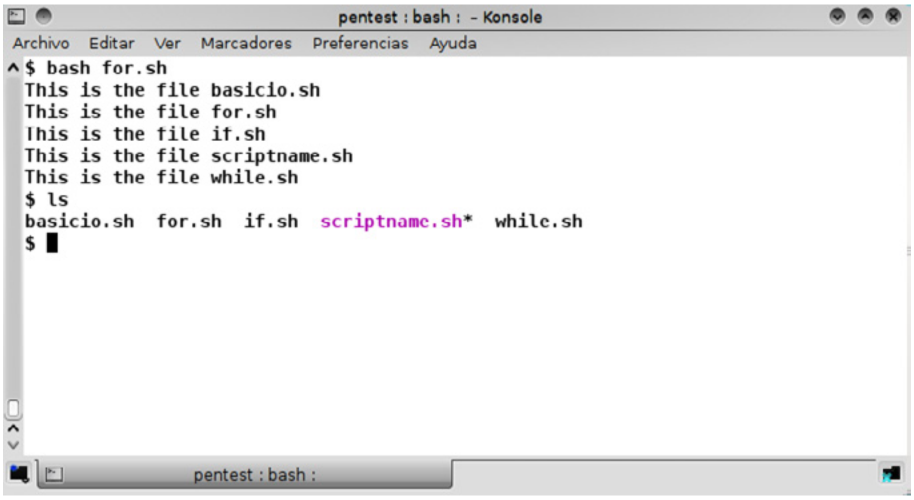


Figure 8. Controlling the flow with the FOR statement

Figure 8 shows the script execution. the command *$(ls .)* listed all the files within the current directory, and used this information to generate a list that was provided for the *FOR* statement. In each iteration the variable *$file* picked a different file from this list, and this was displayed to the user. Once this variable picked all the files within this list, the *FOR* statement stopped its execution

The last flow control that we will see is the *CASE* statement. This function works as several *IF* statements. The function evaluates a variable, and depending on the value, takes different courses of action. The last option * is true only when all the other options are false. The proper syntax is presented in Listing 3.

Listing 3. CASE statement

```
case $VARIABLE in
option1 )
   Command1
   ;;
option2 )
   Command2
   ;;
option3 )
   Command3
   ;;
* )
   Command4

   ;;
esac
```

Suppose that we need to perform different actions depending on a value typed by the user. The code will be like the one in Listing 4.

Listing 4. CASE statement – performing actions depending on the user's behavior

```
#!/bin/bash
echo -n "Select a number from 1 to 3"
read number
case $number in
1 )
   echo "Number 1 was selected"
   ;;
2 )
   echo "Number 2 was selected"
   ;;
3 )
   echo "Number 3 was selected"
   ;;
* )
   echo "Wrong number selected"
   ;;
esac
```

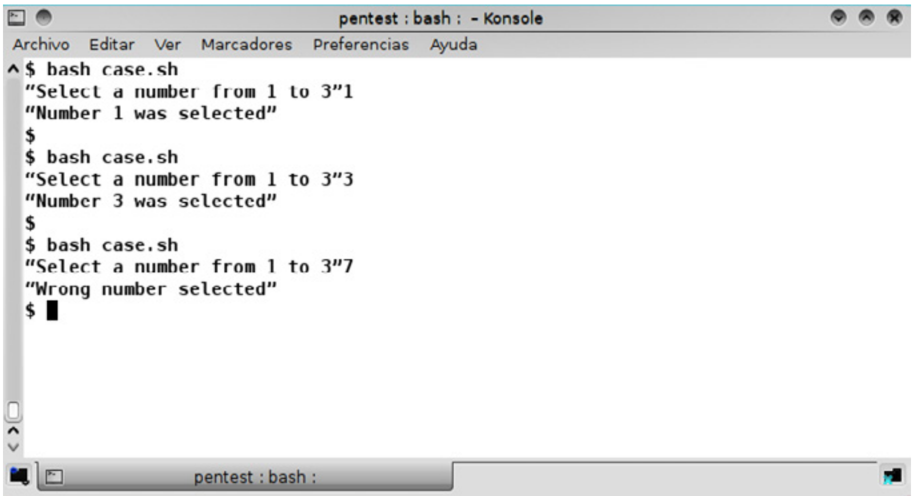


Figure 9. Controlling the flow with the CASE statement

Figure 9 displays three executions of the same script. In the first one, the *CASE* statement evaluated the value of the variable *$number*, and decided to for through the first option. In the second scenario, tool is the third option; and in the last one, as not other option was true, the *CASE* statement went through the * option, the default one.

System Variables

There are a number of system variables that may be helpful when coding shell scripts. These variables are automatically generated by the shell, and could be invoked by the user. Below is a list of the most importants ones.

- *$0*: this variable represents the name of the script, or command that is being executed. If we are executing a script called *test.sh*, the variable *$0* will contain this string
- *$1, $2,...,$n*: these variables are the parameters received by the script. In the previous examples, we requested a user input by using the read function; this could be avoided by allowing the user to execute the script and passing the input as script parameters. For example, if we execute an script like *test.sh string1 number1 variable2*, we are using three parameters, that are being numerated *$1*, *$2*, and *$3* respectively
- *$#*: this variable holds the amount of parameters used when the script was executed. This variable is important if we need to ensure that the script is being executed with a certain amount of parameters

- *$?*: this variable holds the exit code of the last executed command. In Unix/Linux, every command executed, no matter if this was successfully executed or not, returns an exit code. If this code is *0*, then it means that the command executed successfully; however, if this code is different than *0*, this means that there was an error during the execution. Most of the commands have several exit codes to easily identify which was the error that did not allowed the command to complete the required task

- *$**: this variable holds all the parameters used during the script execution. The parameters are usually separated with blank spaces. Considering a script execution with three parameters, the variable *$** will be similar to *$1 $2 $3*

- *$_*: this variable contains the last executed command. Note that if the command executed contained blank spaces, this variable will hold the last parameter

- *$$*: this variable contains the process number of the current shell. If it is called from a shell script, it will contain the process ID under which it is being executed

- *$USER*: this variable contains the name of the current user that is being used. It is similar to executing the command *whoami*

- *$HOSTNAME*: this variable contains the name of the host

- *$PATH*: this variable contains all the paths on where the shell will try to find the command to execute. For example, when the *ls* command is executed, the shell will search in all the paths holded in this variable until the command is found. If it is not found, then an error is presented to the user. If a command or script that needs to be executed is not in any of the paths within this variable, then the full path should be specified, for example, by executing the script as */root/pentest/script.sh*

- *$PWD*: this variable contains the current working directory. It changes every time the *cd* command is invoked

Below is a script example that will validate the amount of parameters received, and will display those to the user (Figure 10)

```
#!/bin/bash
echo "The script was executed with $# parameters"
echo "The first parameter is: $1"
echo "The third parameter is: $3"
echo "All the parameters are: $*"
```

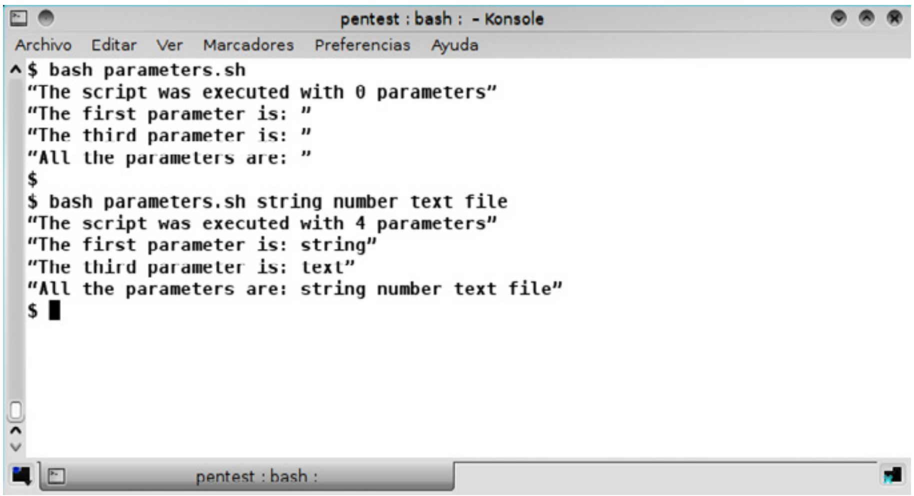


Figure 10. Parameter evaluation

Shell Operators

Besides the system variables, there are operators that work at the shell level, allowing to have a more defined control of the execution of scripts and commands. In Table 2, there is a short description of the most important ones.

Table 2. Most important shell operators

Operator	Effect
.||	This operator could be observed as a logical OR, using the exit codes of the commands. When this operator is used between two commands, it will only execute one of them. If the first command is executed successfully, then the second command will not be executed, as we only requested one of both. If the first command failed for any reason, then the second command will be executed
&&	This operator could be observed as a logical AND, using the exit codes of the commands. When this operator is between two different commands, the shell will try to execute both. If the first command executed successfully, then the second command is also executed. If the first command failed for any reason, then the second command will not be executed, as in this case we requested both commands to be executed successfully
;	Similar to the logical AND and OR, the semicolon character is used within two different commands. However, in this case, there is no relation between them. It instructs the shell to execute the first command, and no matter what was the exit code, the second command will be executed anyway
\	This operator is related to the metacharacters that the shell interprets. The shell has some metacharacters that will interpret, instead of displaying these as strings. For example, some metacharacters are <, >, and $. If these characters are needed as text, the shell should be instructed to not interpret these as metacharacters. This could be performed by using the backslash "\". This character disables the metacharacter interpretation for the next character only, and allows this to be used as text
*	This metacharacter represents from zero to many repetitions of any character. It is a wildcard that could be used to match any missing characters
?	This metacharacter is also a wildcard that could be used in the shell. However, this character represents zero or at least one character
'	The single quotes are used to englobe text within them. Every character within two single quotes will be interpreted as text, no matter what metacharacters are between them
"	The double quotes acts similar to single quotes, except that there is a variable substitution. This means that the variables are being interpreted as that. Also, the backslash is interpreted by the shell
`	The back quotes allows to perform the execution of the command between this quotes, before the result is used. The same result could also be obtained by using "$(command)"

The difference between *'* and *"* is tricky, so probably an example will help to understand this. Figure 11 displays what are the differences between single quotes and double quotes, using the *echo* command, and most of the metacharacters that the shell provides. It also displays how the back quotes actually works.

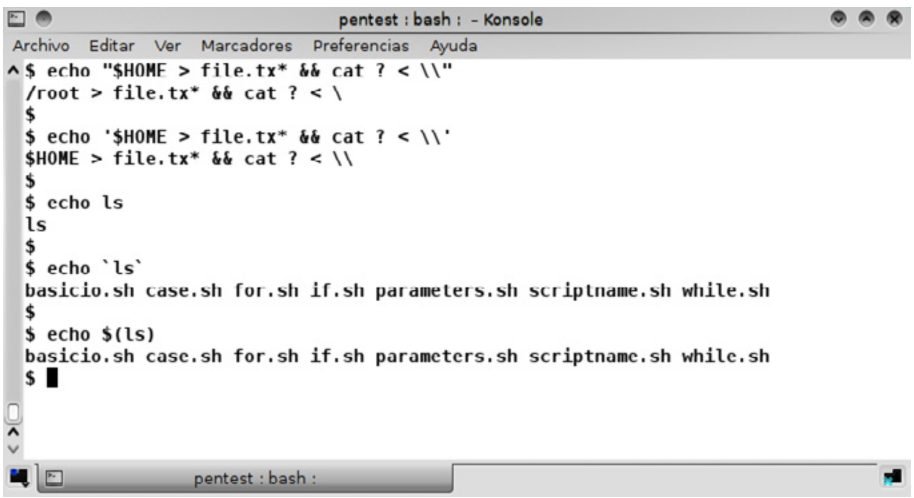


Figure 11. Single quotes, double quotes and back quotes

Regular Expressions

The shell also provides the alternative to use regular expressions, which are sequences of characters that form a pattern used in pattern matching. The regular expressions are basically used when part of the string to match is known, and part is not know. The characters involved in a regular expression could be metacharacters or a regular character. The metacharacters represents characters with a special meaning, similar to those mentioned earlier (*, $, \, etc.); and the regular characters are those that form the string, those characters with no special meaning, or a meaning that was annulled using the backslash. The list in Table 3 contains the metacharacters that are involved in a regular expression.

Table 3. Metacharacters involved in a regular expression

Operator	Effect
.	It matches any character, but just one occurrence
?	Indicates that the preceding item is optional, and if matched, will match only one character
*	Indicates that the preceding item is optional, and if matched, will match up to several characters
+	Indicates that the preceding item will be matched one or more times
{N}	Indicates that the preceding item is matched exactly N times
{N,}	Indicates that the preceding item is matched N or more times
{N,M}	Indicates that the preceding item is matched at least N times, but not more than M times
[A-Z0-9]	It matches every uppercase letter or number, on time only. It could also be used with one range only, or even with a predefined characters instead of a range
\	Similar to the bash metacharacters, the backslash escapes the special interpretation of the next character
(reg1|reg2)	These characters are used to provide alternatives to the regular expression matching. It will use reg1 or reg2 to matching, but not both at the same time
-	Represents a range if it's not first or last character in a list
^	It matches the beginning of a line
$	It matches the end of a line. In regular expressions, this character has a different meaning that when we work with variables
^$	These two characters used in conjunction indicate an empty line, as this regular expression will match the beginning of the line, followed by the end of the line
\<	It matches the empty string at the beginning of a word
\>	It matches the empty string at the end of a word

The square brackets also provide an alternative way to specify ranges. Table 4 presents a list of the predefined values that could be used within this regular expression.

Table 4. Predefined values used in regular expressions

Operator	Effect
[:alnum:]	It matches alphabetic or numeric characters. This is the equivalent to the range [a-zA-Z0-9]
[:alpha:]	It matches alphabetic characters. This is the equivalent to the range [a-zA-Z]
[:blank:]	It matches a space or a tab characters
[:cntrl:]	It matches control characters
[:digit:]	It matches decimal digits. This is the equivalent to the range [0-9]
[:graph:]	It matches printable characters. This is the same as [:print:] but excludes the space character
[:lower:]	It matches lowercase alphabetic characters. This is the equivalent to the range [a-z]
[:print:]	It matches printable characters. This is the same as [:graph:] but includes the space character
[:space:]	It matches blank spaces characters
[:upper:]	It matches uppercase alphabetic characters. This is the equivalent to the range [A-Z]
[:xdigit:]	It matches hexadecimal digits. This is the equivalent to the range [0-9a-zA-Z]

Below are some examples of regular expressions and a short explanation of how this is achieved:

- _ls -l [a-cx-z]*_: This command lists, using the long format, all the files in the current directory whose names start with a, b, c, x, y or z. Note that this is case sensitive, and will not list files that starts with these letters in uppercase

- _grep [[:digit:]] file.txt_: This command will show the lines that contain digits within the file _file.txt_

- _ls -l [[:upper:]]*_: This command lists, using the long format, all the files in the current directory whose names start with an uppercase letter.

- _grep ^root /etc/passwd_: This command will show the line that starts with _root_ within the file _/etc/passwd_

- _grep [abc] /etc/group_: This command will show the lines that contain the letters a, b or c within the file _/etc/group_

- _egrep “(a|b|c)” /etc/group_: This command is similar to the previous one, but instead of using square brackets with a predefined range, it is using the pipe

- _grep :$ /etc/passwd_: This command will show all the lines that end with _:_ within the file _/etc/passwd_

- _grep ‘\<b..h\>’_ *: This command will show all the lines that contain a word that starts with b, followed by two characters, and ends with an h.

- _grep -v ^$ file1.txt >file2.txt_: This command will remove all the empty lines within the file _file1.txt_, and will save the results into the file _file2.txt_

There are more examples of regular expressions available, but this section described the most common uses, and could be adapted according to the situation.

Useful Commands

We have learned the internal structure of a shell script, the system variables, and how to control the data flow. This is important when trying to write a shell script; however, this is not all the necessary knowledge to write scripts. The other important part of shell scripting is to known the Unix/Linux commands that could be used. This is a list of the most common commands that are usually used in shell scripting.

- _cat_: this command display the content of the files that is passed as a parameter. An * wildcard could be used to display the content of all the files that are in the current working directory

- _cut_: this command cuts the standard input received in columns, used a predefined character to separate each column from the others. For example, if we are working with a CSV file, the character that will be used to cut the file in columns is the comma

- _grep_: this command will search for a specific string within the standard input provided. This command is useful to find which text file contains a specific string. This command is case sensitive by default, but this could be changed with the parameter _-i_

- _awk_: this command is used for processing column-oriented text data, such as tables, presented to it on standard input. The parameters are the contents of the columns in the current input line

- _find_: this command conducts a search for a particular file or file properties within the specified directory and subdirectories

- _date_: this command displays the current date, time, and timezone of the host

- _tar_: this command creates a container which stores all the files that were received as standard input. It is similar to creating a zip file, but _tar_ does not perform any type of compression

- *wc*: this command counts the amount of lines, words, and bytes respectively within a text file provided by standard input
- *tr*: this command could be used to replace one string with another. It could also be used to remove certain strings, or even delete duplicates of a particular string
- *xargs*: this command reads the standard input, which is delimited by blank spaces or newlines, and then execute a command to each of these new lines
- *sed*: this command is a powerful string editor that could be used to replace, and/or remove text from the standard input. It works by using regular expressions

sed and *awk* are perhaps the most important tools to parse string blocks and files. They also are the most complex tools to use, and will require a full article to learn all the capabilities that these tools provides. Below, there are a couple of examples that could help to understand the logic of these tools.

- *sed s/day/night/ <input.txt >output.txt*: The *s* parameters indicates that *sed* will replace the first occurrence of the string *day* in each line of the *input.txt* file with the string *night* and will save the modifications into the file *output.txt*
- *sed s/day/night/g <input.txt >output.txt*: This command is similar to the previous one, except that the *sed* parameters end with a *g*. This indicates to *sed* to perform the changes globally. This will change all the occurrences of the word *day* with the word *night*
- *sed 's/[0-9]*/& &/' <file.txt*: This command will match strings that are only numbers, and will add a second string similar to the one matched. This is performed by using the & character
- *awk '{print $2,$3;}' file.txt*: This command will display only the columns 2 and 3 of *file.txt*. By default, the columns should be separated with a blank space.
- *awk -F ':' '$3 >200' /etc/passwd*: This command will show only lines where the third column is greater than 100. The *-F* parameter is used the define the delimiter within the different columns

There are several more commands we could talk about, but are more related to Unix/Linux administration, rather than shell scripting. Shell scripting is a powerful tool that allows the automation of most of the system administration tasks, and could be used to perform several penetration testing tasks. Now it is up to you to take advantage of this and start building your own scripts. As most of the programing and scripting languages, the best way to learn it is by coding.

About the Author

Sebastian Perez, C|EH, CISSP ar.linkedin.com/in/sebasperez/
Sebastian is a Senior Security Consultant in one of the Big Four, located in Buenos Aires, with an experience of more than seven years providing IT Security services. He is responsible for delivering security solutions; including penetration testing, vulnerability assessment, security audits, security remediation, mobile and web application penetration testing, infrastructure security assessments, network analysis, as well as systems architecture design, review, and implementation.
He also offers internal trainings related to ATM security and penetration testing Android apps. Prior to joining his current company, he worked as a systems administrator, security policy manager and IT security consultant. His background gives him a thorough understanding of security controls and their specific weaknesses. Furthermore, he is proficient in multiple security application tools, network technologies and operating systems. Sebastian has a post-degree in Information Security, and is currently working on his master degree thesis related to computer and mobile forensics. He also published a CVE # CVE-2012-4991 related to the Axway Secure Transport software, which was vulnerable to Path Traversal vulnerability.

Job Management

by Steven Poulsen

The Linux terminal provides access to a powerful shell along with an abundance of powerful Linux commands. A small piece of this is the power in job management. This article will explore several aspects of the Linux "job," which will allow the reader to control Linux processes in more time-saving ways.

In this article, we divide the job management into three categories: immediate, delayed, and scheduled (Figure 1). These categories define how the various commands are used in day to day processes. Not only do they deal with the time aspect of jobs, but also reusability and effort (Figure 2).

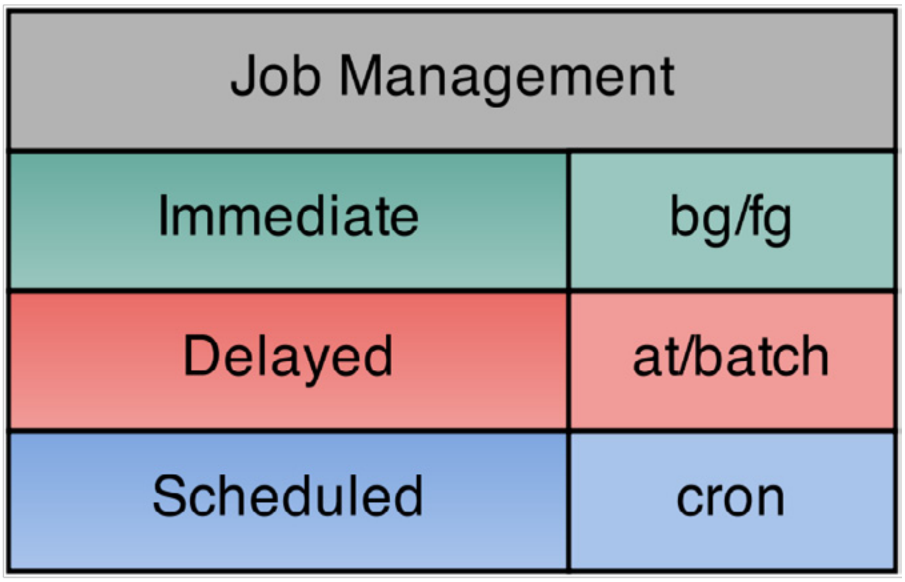


Figure 1. Job management categories

Basic/Immediate Job Handling

The standard Linux shells provide job management that can be used to increase efficiency. As we work in the shell, we often are performing several tasks, some of which are dependent upon others, but many that are not. We may run a command to perform a task and then find that it is taking too long and we could be doing something else. We may be tempted to open another terminal and work with other tasks, but this is not necessary and often harder to manage. Quite often, we simply desire a way to put a task into the background to give us access to working on other tasks. For example, we may wish to rsync some files to another server.

Let's issue a command similar to the following:

```
$ rsync -a Documents carbuncle:backup/
```

…and so we wait, and wait, and wait. After about 15 seconds, I often find the needs to do one of two things: a) kill the command with CTRL-C and then run it again with the -av option to see what is taking so long, or b) put it in the background.

To put it in the background, it is important to not start the command with -v so that rsync does not generate lots of output. It is hard to work in a shell with output streaming by.

Press CTRL-Z.

You will then see a line like the following:

```
[1]+  Stopped                 rsync -a Documents carbuncle:backup/
```

Followed by a prompt. The job is suspended, and the prompt is returned. Normally, the next thing most users will do is to issue the “bg” command. However, we will investigate things a bit more before doing so.

Issue the following:

```
$ jobs
```

This command will show the jobs and their current state.

```
[1]+  Stopped                 rsync -a Documents carbuncle:backup/
```

This is the same line we saw when we pressed CTRL-Z. What it shows us is that we have a job #1 that is Stopped and then we can see the job’s command line. What about the +character? This shows us the default job or “current” job. In short, any job commands we execute will be applied to this job by default. Let’s get another job going:

```
$ while [ 1 ] ; do sleep 10 ; done &
```

It is important that you put spaces in the proper places inside the brackets. The above command is replicated with a + to show the important spaces:

```
$ while+[+1+]; do sleep 10 ; done &
```

When you run this command, you should see the following output:

```
[2] 75774
```

What we get here is the job number 2 and the process ID 75774. Type “jobs” again to see what is running.

```
[1]+  Stopped                 rsync -a Documents carbuncle:backup/
[2]   Running                 while [ 1 ]; do
    sleep 10;
done &
```

This time around, we used the ampersand (&) at the end of the line to force it to run in the background. Now we have a job running in the background that runs forever. Let’s repeat the above command without the ampersand:

```
$ while [ 1 ] ; do sleep 10 ; done
```

Now press CTRL-Z:

```
[3]+  Stopped                 sleep 10
```

Then type “jobs”

```
[1]-  Stopped                 rsync -a Documents carbuncle:backup/
[2]   Running                 while [ 1 ]; do
    sleep 10;
done &
[3]+  Stopped                 sleep 10
```

We now have another case where we have three jobs that are being manages, at various states. We also now have a minus symbol next to the first job’s identifier: [1]-. This identifies the job that will become default, when the default job no longer exists or is running. Let’s start job #3 so that it can run the same as job #2:

```
$ bg
```

This command will put the default job [3] into a Running state.

```
[1]+  Stopped                 rsync -a Documents carbuncle:backup/
[2]   Running                 while [ 1 ]; do
    sleep 10;
done &
[3]-  Running                 while [ 1 ]; do
    sleep 10;
done &
```

Job #2 and #3 are now both running similarly and Job #1 has become the default. Now we look at killing the first job:

```
$ kill %1
Killed by signal 15.
rsync error: unexplained error (code 255) at /SourceCache/rsync/rsync-42/rsync/rsync.c(244)
[sender=2.6.9]
```

Then viewing the jobs again with the “jobs” command:

```
[1]+  Exit 255                rsync -a Documents carbuncle:backup/
[2]   Running                 while [ 1 ]; do
    sleep 10;
done &
[3]-  Running                 while [ 1 ]; do
    sleep 10;
done &
```

Let’s issue the “jobs” command one more time:

```
[2]-  Running                 while [ 1 ]; do
    sleep 10;
done &
[3]+  Running                 while [ 1 ]; do
    sleep 10;
done &
```

Notice what happened. After a job changes state, it will show up in the job list one final time to show us what happened. We can see that job #1 exited with exit code 255. However, when we issue “jobs” a second time, we see that we now have two jobs running in the background. How do we know they are running in the background? We know this simply because we have a terminal prompt and are able to issue the “jobs” command.

While these jobs run, we can choose to bring one to the foreground. We can simply type “fg” to bring the default job to the foreground, but we may want to bring a different one. We have two options:

```
$ %2
```

Or

```
$ fg 2
```

Both of these commands will bring job #2 to the foreground. When yo type these commands, you will see the command printed, but will no longer have a prompt:

```
while [ 1 ]; do
    sleep 10;
done
```

You can choose to put this command back into the background with CTRL-Z followed by “fg” or you can simply kill it now with a CTRL-C. Let us do the latter, then issue “jobs” one more time:

```
[3]+  Running                 while [ 1 ]; do
    sleep 10;
done &
```

Because we killed the job in the foreground, we no longer see its exit code in the job list. Let’s try another job:

```
$ sleep 5 &
[4] 76079
```

We now wait over 5 seconds and then issue the “jobs” command once again:

```
[3]-  Running                 while [ 1 ]; do
    sleep 10;
done &
[4]+  Done                    sleep 5
```

We can see that job exited nicely by the Done status. Let’s kill all jobs by issuing “fg” followed by CTRL-C until no more jobs exist. If you now type “jobs” you will see no jobs are running.

One consideration while running jobs is what happens with their output and input. Let’s first look at the output by running a fake job:

```
$ while [ 1 ] ; do sleep 2 ; echo Tick ; done &
```

You will notice that you get your prompt back, but are getting annoying output that interrupts any further typing.

```
$ Tick
Tick
Tick
Tick
Tick
Tick
```

When you run a command in the background that generates output, you have a few options. One of them is to eliminate the output by redirecting it to /dev/null

```
$ while [ 1 ] ; do sleep 2 ; echo Tick ; done 2>&1 > /dev/null &
```

The problem with this method is that you may want to view it later. If that is the case, another option is to redirect the output to a file:

```
$ while [ 1 ] ; do sleep 2 ; echo Tick ; done 2>&1 > /tmp/mylogfile &
```

This method allows one to “cat” or “tail” the file as needed to check on progress. A third method to handle the output is to consider “batch job processing” which is covered in the next section.

Before we conclude with the basics, you may be wondering still about script input. What happens when a job in the background needs user input? Let’s investigate by using the “read” command to ask for input:

```
$ ( sleep 2 ; read -p "Enter something" ) &
[1] 76541
```

Wait a couple seconds until you see the “Enter something” prompt, then press enter at the prompt. The following will show up to indicate the job has stopped:

```
[1]+  Stopped                 ( sleep 2; read )
```

It would appear that the job is stopped and simply needs to be put in the background. Issue "bg":

```
[1]+  Stopped                 ( sleep 2; read )
```

It refuses to continue so issue another "fg" command to bring it to the front, then enter the input; any text will do. The job will then exit.

In summary, a command that needs input will stop running and wait for it to be put in the foreground.

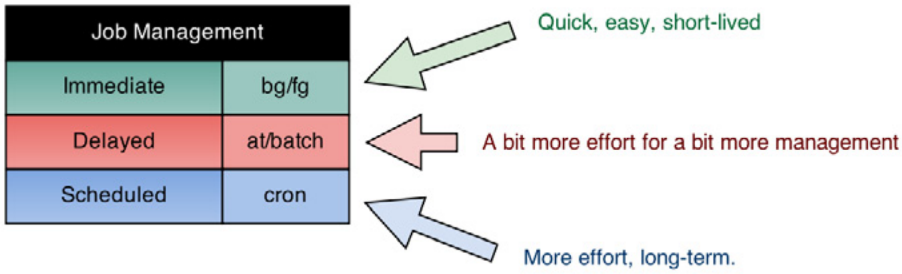


Figure 2. Effort

Delayed (Batch) Job Handling

Linux systems generally provide command for batch job processing. These commands are geared toward the processing of commands at a later time or as system load is low.

Let's first start with the batch command. This command will allow the background execution of a command and differs from the simple "bg" or "&" methods in a few ways. Try this example:

```
$ batch
```

When this command is issued, you will be taken to a new prompt:

```
at>
```

Here we will enter some commands to run, followed by CTRL-D

```
at> sleep 10
at> echo Testing
at> CTRL-D
```

You should then see the following output:

```
job 10 at Sun Nov 24 17:41:00 2013
```

Afterward, the job will run and if your machine is properly setup, you should receive mail with the output:

```
Return-Path: <spoulsen@signetik.com>
X-Original-To: spoulsen
Delivered-To: spoulsen@signetik.com
Received: by mail.signetik.com (Postfix, from userid 1000)
   id 053A0FF03C; Sun, 24 Nov 2013 17:54:20 -0600 (CST)
Subject: Output from your job       10
To: spoulsen@signetik.com
Message-Id: <20131124235421.053A0FF03C@mail.signetik.com>
Date: Sun, 24 Nov 2013 17:54:20 -0600 (CST)
From: spoulsen@signetik.com (Steve Poulsen)
Testing
```

We will now create a new command that will take much longer to complete, to give us time to investigate. Issue “batch” then the following commands:

```
at> sleep 3000
at> echo Testing
at> <EOT>
job 11 at Sun Nov 24 17:57:00 2013
```

Now issue the “atq” command to view the “at queue”:

```
11     Sun Nov 24 17:57:00 2013 = spoulsen
```

From this command, we get the list of jobs that are in the queue. We get the ID (11), the date, the user name (spoulsen), and the actual queue that this job is attached to (=). We will look at different queues later, but for now keep in mind that the “=“ is the queue of jobs that are running.

We will leave that job running and add another job scheduled to run in one hour. In order to queue a job to run at a later time, we use the “at” command:

```
$ at now + 1 hour
```

We are now presented with a similar interface as the batch command. Again, enter the commands followed by CTRL-D

```
at> sleep 4000
at> echo at testing
at> <EOT>
job 12 at Sun Nov 24 19:03:00 2013
```

Once again list the jobs in the queue with “atq”:

```
$ atq
12     Sun Nov 24 19:03:00 2013 a spoulsen
11     Sun Nov 24 17:57:00 2013 = spoulsen
```

We can see we have another job in queue “a”, which is the default for “at” commands. Queue “a” is where “at” commands are held until it is time to run. Queue “b” is where “batch” commands are held until they run. Other queues (c-z) can be used and simply increase the niceness, or decrease the load they put on the system, when they run. Queues (A-Z) work the same as the lower case queues, but instead hand off the job, at the set time, to the “batch” system so that they further wait for the system load to be low before running.

Let’s now add another job that will run soon:

```
$ at now + 1 minute
at> sleep 30
at> echo “30 second job complete”
at> <EOT>
job 13 at Sun Nov 24 18:11:00 2013

$ atq
13 Sun Nov 24 18:06:00 2013 a spoulsen
12 Sun Nov 24 19:03:00 2013 a spoulsen
11 Sun Nov 24 17:57:00 2013 = spoulsen
```

We see that we have our original batch job [11] running and two “at” jobs that are scheduled to run at 19:03 and 18:06. In just a minute, when 18:06 is reached, the job runs, and the email comes:

```
Return-Path: <spoulsen@signetik.com>
X-Original-To: spoulsen
```

```
Delivered-To: spoulsen@signetik.com
Received: by mail.signetik.com (Postfix, from userid 1000)
   id C2BE4FF03C; Sun, 24 Nov 2013 18:14:10 -0600 (CST)
Subject: Output from your job      13
To: spoulsen@signetik.com
Message-Id: <20131125001410.C2BE4FF03C@mail.signetik.com>
Date: Sun, 24 Nov 2013 18:14:10 -0600 (CST)
From: spoulsen@signetik.com (Steve Poulsen)

30 second job complete
```

Now examine the jobs again:

```
$ atq
12 Sun Nov 24 19:03:00 2013 a spoulsen
11 Sun Nov 24 17:57:00 2013 = spoulsen
```

Let's kill the jobs and call it good. In order to do this, we need to use one more command "atrm":

```
$ atrm 11
Warning: deleting running job
$ atrm 12
```

Scheduling Jobs

In this section, we introduce the "cron" utility. Cron takes the idea of batch job processing to next level by allowing jobs to be scheduled in a more permanent way. Most Linux systems have a version of cron installed and running, or at least a package that can install it. In order to verify cron is installed, use "whereis":

```
$ whereis cron
cron: /usr/sbin/cron /etc/cron.monthly /etc/cron.hourly /etc/cron.d /etc/cron.daily /etc/cron.
weekly /usr/share/man/man8/cron.8.gz
```

Cron allows users and the administrator to permanently configure jobs to run at various times. There are two methods in getting cron to run a job. We will first look at the user level crontab files, which indicate the jobs along with their scheduling information. To setup a new job, we use "crontab -e" to edit the crontab files.

The default crontab file generally has comments at the top to remind the user how to add new entries. Entries are single lines that contain the schedule and the task. Lines are entered in the following format:

```
m h  dom mon dow   command
```

Where, m is minute, h is hour, dom is day of the month, mon is month, dow is the day of the week and command is the command to run. Each of the entries, except for command, is best understood by first knowing that an asterisk can be used to indicate "all values". For example:

```
* * * * * /bin/ls ~
```

This entry means to run on all days of the week, all months, all days of the month, all hours and all minutes. This causes this command to run every minute. Therefore, every minute, the command runs and the output is mailed to the current user, or in other words, the user that edited the crontab. It is important to understand that all users have their own crontab file and their commands are their own and run as their user account, with appropriate permissions.

There is not much more to say about the crontab file, other than the fact that the time values can be quite complex. For example:

```
0-59/15 0,6,12,18 1 * * /opt/mailbackup/bin/go
```

This example demonstrates several features of the time system. The minutes value of 0-59/15 means to run for all minute values between 0 and 59 that are divisible by 15, perfectly. Therefore, this command will run at 0 minutes, 15 minutes, 30 minutes, and 45 minutes past the hour. We obtain the hours from the next argument, which is simply a list of hours to run this command. Finally, we further qualify this cron entry by the day of the month, which is 1. So, on the 1st of every month, at midnight, 6 a.m., noon, and 6 p.m., this command will 4 times during that hour. Specifically, at 0:00, 0:15, 0:30, 0:45, 6:00, 6:15, 6:30, 6:45, 12:00, 12:15, 12:30, 12:45, 18:00, 18:15, 18:30, and 18:45, on the 1st and only on the 1st of the month, this command runs.

Cron also allows for application specific files to exist in `/etc/cron.d` using the same format. This is really no different than the user files, other than the fact that a package can easily install and uninstall single files.

Finally, cron allows for application specific scripts to exist in `/etc/cron.hourly`, daily, weekly, and monthly. The files in these folders are simply scripts that cron executes at the matching interval.

Scripting

Dealing with jobs can be quite painful without the use of scripts. Sometimes we start jobs that are one-liners, but at other times we want more advance methods to occur. For example, consider a mail back script:

```
 1 | #!/bin/bash -e
 2 |
 3 | DATE=`date +%Y%m%d`
 4 | DIR=mail.backup.${DATE}
 5 | /usr/bin/rsync -a Maildir/ ${DIR}
 6 |
 7 | ALL=${1}
 8 | HOST=`hostname`
 9 |
10 | if [ “${ALL}” != “all” ] ; then
11 |  find Maildir -mtime +180 -type f -name \*.${HOST}\* -exec rm {} \;
12 |  find ${DIR} -mtime -180 -type f -name \*.${HOST}\* -exec rm {} \;
13 | fi
14 |
15 | tar cvfj ${DIR}.tbz2 ${DIR}
```

Lines 3-4 setup the backup mail directory based on the current date

Line 5 copies the current mail directory to the backup directory

Lines 7-9 sets some more variables used in the backup

Lines 10-13 will remove old files form the Maildir and remove new files from the backup directory

Line 15 tars the backup directory, which now contains only files that are 180 days old, or older.

WE would like this command to run once a month, on the first of the month. Because we want a periodic command to run, we choose to use cron:

```
$ crontab -e
45 4 1 * * ( cd /home/spoulsen && /opt/bin/email_backup.sh )
```

This is now setup to run on the 1st of every month, at 4:45 a.m. We can forget about this from now on and on the 1st of the month, it will run and then email us the resulting output. When we receive this email, we examine it for errors, then delete it and move on to more important tasks.

Concurrency

In the previous scripted example of a mail backup system, things were pretty safe. However, what would happen if the script took longer than a month to run? We would end up with multiple copies of the script running, both of them trying to copy and delete files in the same directories. This could be disastrous.

Of course, in the mail backup example, this would never happen since the script generally finishes in just a few minutes. However there are cases where this really can become a problem. We often will use cron to schedule jobs that keep folders in sync and they might run once per hour and could take several hours to complete. For these cases, we need to protect against concurrently running scripts. Let’s review the following script additions.

```
 1 | scriptlock=/tmp/.batch.lck
 2 |
 3 | rmlock() {
 4 |     rmdir ${scriptlock}
 5 | }
 6 |
 7 | fexit() {
 8 |     rmlock
 9 |     echo “Finished the mail backup”
10 | }
11 |
12 | mkdir ${scriptlock} 2>/dev/null || exit 1
13 |
14 | trap ‘fexit’ EXIT
15 |
16 | echo “Starting the mail backup”
17 |
18 | # The meat of the script follows
```

Line 1 sets the lock file name

Lines 3-5 define a function to remove the lock file

Lines 7-10 define an exit function that removes the lock file and print the exit string

Line 12 creates the lock file

Line 14 sets up an EXIT trap

Line 16 prints the start string

You may be wondering why we use the “mkdir” command to create a lock file. The reason is because this operation is atomic. If it succeeds, it creates the directory and returns success. Otherwise, it returns an error. It would be sufficient to create the directory and remove it in a script, but what happens if the scripts errors or is killed? If we want to ensure the lock file is removed on any exit, we need to “trap” the EXIT signal with a script function. This way, when the script exists by any means, our fexit() function is called and will remove the lock.

Logging

While we are at it, let’s get a little better with our script logging. Linux systems generally have a syslog daemon that runs and collects messages from all sorts of processes. These messages are filtered and placed in various files under `/var/log`. The filtering of these messages is handled by either syslog.conf, rsyslog.conf, or other syslog flavors’ config files. See Figure 3.

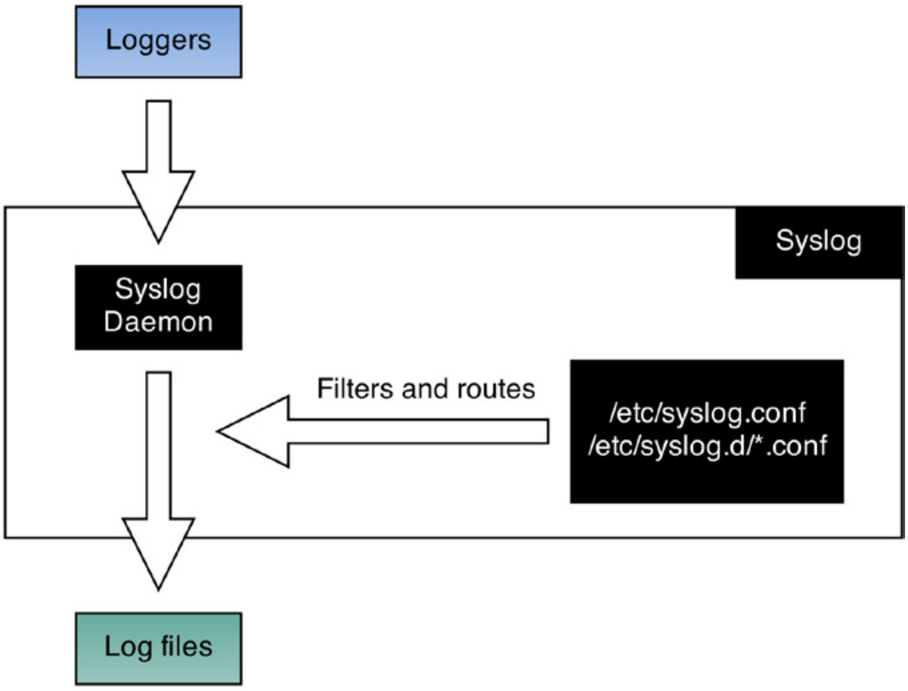


Figure 3. Logging

Our scripted jobs can take advantage of the syslog system by use of the "logger" command. Let's try out a few command and see what happens.

```
$ logger "My message"
$ tail /var/log/syslog
Nov 24 20:03:48 carbuncle dhcpd: DHCPACK on 10.55.10.238 to c8:d1:0b:19:ec:2a (Windows-Phone) via
   eth0
Nov 24 20:04:19 carbuncle dhcpd: DHCPDISCOVER from 58:67:1a:72:6e:49 via eth0
Nov 24 20:04:20 carbuncle dhcpd: DHCPOFFER on 10.55.10.214 to 58:67:1a:72:6e:49 via eth0
Nov 24 20:04:20 carbuncle dhcpd: DHCPREQUEST for 10.55.10.214 (10.55.10.9) from 58:67:1a:72:6e:49
   via eth0
Nov 24 20:04:20 carbuncle dhcpd: DHCPACK on 10.55.10.214 to 58:67:1a:72:6e:49 via eth0
Nov 24 20:04:28 carbuncle spoulsen: My message
```

Syslog utilizes a facility.level nomenclature for its messages. The facility says where the message came from and the level states how important it is. You can look these up with the man page of logger:

```
     Valid facility names are: auth, authpriv (for security information of a
     sensitive nature), cron, daemon, ftp, kern (can't be generated from user
     process), lpr, mail, news, security (deprecated synonym for auth), sys     log, user, uucp,
   and local0 to local7, inclusive.

     Valid level names are: alert, crit, debug, emerg, err, error (deprecated
     synonym for err), info, notice, panic (deprecated synonym for emerg),
     warning, warn (deprecated synonym for warning).  For the priority order
     and intended purposes of these levels, see syslog(3).
```

So, we can retry our logging:

```
$ logger -p user.emerg "Error"
```

However, this may not do anything different, unless we have our syslog config file set appropriately. In order to do this, find the syslog.conf file to change. If your system uses rsyslog, you would find it in rsyslog.d/50-default.conf. We are looking to enable or add the following line.

```
user.*                          -/var/log/user.log
```

Restart the rsyslog server:

```
$ sudo service rsyslog restart
rsyslog stop/waiting
rsyslog start/running, process 22898
```

Now repeat the logger command:

```
$ logger -p user.emerg "Error"
[10] $ tail /var/log/user.log
Nov 24 20:27:31 carbuncle spoulsen: Error
```

Now let's use the -t (tag) option to get rid of our name and put in a more useful tag.

```
$ logger -p user.emerg -t backup "Error"
$ tail /var/log/user.log
Nov 24 20:27:31 carbuncle spoulsen: Error
Nov 24 20:28:10 carbuncle backup: Error
```

Conclusion

In this article, we started with basic job management and progressed into more powerful scripted batch processing to allow complex tasks to be performed. In the simple subject of job management, there exists many feature rich Linux tools to help us process tedious tasks in more efficient ways. These tools help us to free up our time by forcing the jobs to wait on us, rather than us waiting on the jobs. The next step for the reader is to put these tips into practice and to become more efficient in the day to day tasks that demand our expertise.

About the Author

Steve Poulsen is an independent consultant in embedded systems, with an emphasis on embedded Linux and Android. Steve's background is in security systems, digital audio and video processing, embedded Linux design and development, and PC infrastructures to communicate with embedded devices. He currently currently works with Communication Systems Solutions, providing embedded design services and IT support.

Forticom

Prevent fraud at the front line rather than detecting it after it has happened

Forticom introduces a ground-breaking identity validation and authentication solution that effectively eliminates the potential for credentials to be stolen and subsequently misused. Our solution can be quickly and easily integrated into most existing systems and processes without the need for complex changes to you or your users.

- Reputation Protection
- Fraud Prevention
- Unified process across all entry points
- Reduced costs
- Reduced complexity

- No devices required
- Flexible implementation
- Immune to phishing and scamming
- User initiated duress capabilities
- Immune to observation

www.forticomgroup.com

Privacy and Anonymity Techniques Today

by Adrian Lamo

The Internet is a serious business. This wasn't always the case. The phrase, in fact, originated as a form of wry mockery of netizens who took things just a bit too seriously. But, that was then. Today it's indisputable that the Internet is locked in an arms race between privacy and intrusion into that privacy – wholesale intrusion by governments into the lives of others in the name of national security, and enthusiastic intrusions by others into the private affairs of governments under the banner of whistleblowing, both willing and able to pontificate at length about why their brand of snooping is good and proper.

This race can become confusing to anyone not paying encyclopedic attention, and before dwelling on what can be done today, it seems worthwhile on how we got to today from yesteryear.

Just how did it get so complicated?

Whenever it is that the mood of the Internet first grew dark, Robert Tappin Morris' (*http://www.bloomberg.com/slideshow/4/2012-04-18/famous-hackers-then-and-now.html*) sort-of -kind-of-accidental 1988 worm was one of the first significant signs of the Internet's slide from an obscure tool of academia into a place where attack and defense were no longer reserved for video games. The worm managed to take over a significant chunk of the nascent Internet thanks to a programming error which vastly enhanced its rate of replication. Morris' creation was also a contemporary of a milestone in Internet militarization, the case of the Hanover Hackers. Bright young men who offered themselves to the KGB as network spied, they quickly became embroiled in a world perhaps less glamorous than they had imagined. Both were a foreshadowing of things to come.

Still, in 1988 such incidents were remarkable for their rarity – spies? On the Internet? How droll. And yet, it was only 10 years later that the Moonlight Maze (*http://en.wikipedia.org/wiki/Moonlight_Maze*) incident made it clear that computer network exploration was passing out of the hands of hobbyists and into those of professional state actors pumping what was still a young Internet for defense information, who would seek to redefine exploration into computer network exploitation. This had all happened before, just as it would all happen again.

Even if the times were changing, this was no concern to the average netizen. The affairs of nations remained lofty and distant to an Internet whose main problems remained spam, low-grade hacking, and itinerant computer worms. If the average user might crack the occasional joke about ECHELON breathing on the line (*http://news.bbc.co.uk/2/hi/503224.stm*), one who seriously asserted that a three-letter agency was out to nick their e-mail could still be soundly called paranoid. What would the government want with notes from your sewing circle or the occasional Internet exchange of recipes? The Internet had become big business, and with it had Internet crime as well. Fraud and identity theft were the concerns of the day.

And then the turn of the century turned into 2001, and war without end. Chastened and burned by allegations – mostly true – that they had hoarded actionable data out out of a sort of insular inertia, America's intelligence agencies rushed to tack "netcentric" onto everything that didn't already have "terror" nailed onto it. This when they weren't busying themselves with newly-granted expansions of their surveillance powers. (*http://en.wikipedia.org/wiki/Controversial_invocations_of_the_Patriot_Act*). This odd fusion of openness and secrecy would be called many things, but stable would not prove to be one of them.

The turning of the next decade should need no introduction. As it turned out, the feds really *were* after Grandma's recipes, sort of. The recipes and everything else within a ten terabyte radius, at any rate. Whether this was a cause or a consequence of 2010's Wikileaks (*http://www.wired.com/threatlevel/2013/08/bradley-manning-sentenced/*) is open to question – the only thing that's certain is that as netizens growingly adopted radical transparency by any means necessary as their chosen battle, the government had no choice but to escalate in kind. As with most things, the truth probably lies somewhere in between.

And finally in our post-Snowden, post-Manning era, Internet users are coming to realize that they have a stake in this growing hacker cold war. (*http://www.popularmechanics.com/technology/military/news/edward-snowden-vs-bradley-manning-by-the-numbers-15574490*).

The communities they inhabit are caught up in issues of surveillance and related state activities no matter how pedestrian they may have thought themselves. VPN's and other privacy services are no longer the domain of people with something to hide, but that of those who don't believe a desire for privacy against state encroachment *should* be something to hide. As Derek Zimmer at VikingVPN told PenTest "In 2008, a person who ran a VPN was probably downloading movies or defacing websites. Just five years later, everyone is seeing the need to shield themselves from government and corporate intrusion into their lives." If history is any guide, the turn of the next decade will only see this trend advancing.

There are many privacy options out there, and as with the interminable operating system debate (*http://lifehacker.com/mac-vs-windows-your-best-arguments-486125257*) no one's solution is right for everyone. As it is with most people, I can mostly only write what I know: 10 years ago I was a cybercriminal, sought by the FBI, and privacy countermeasures were what kept me free. Today, my work for the government has made me a marked man in the eyes of some, and countermeasures keep me safe. I like to think safety and freedom are all anyone could ask for out of such a service, but I leave it up to you, Gentle Reader, to determine whether the services extant today can accomplish that goal.

TOR

Tor, otherwise known as "The Onion Router" for the onion-like layers of encryption and obfuscation it places between a user and the final object of their Internet usage (*https://www.torproject.org/about/torusers.html.en*), was originally a project sponsored by the US Naval Research Laboratory (*http://en.wikipedia.org/wiki/Tor_%28anonymity_network%29*) in an apparent nod to the fact that private Internet users aren't the only ones with a desire to keep their secret sauce recipe secret – state actors don't like being snooped on (or even identified) any more than the next guy. Indeed, according to a mailing list post by Michael Reed (one of the creators of Tor), the network's raison d'etre was never personal privacy & anonymity, but rather "DoD / Intelligence usage (open source intelligence gathering, covering of forward deployed assets, whatever). Not helping dissidents ... not assisting criminals." (*http://cryptome.org/0003/tor-spy.htm*)

This doesn't mean that the whole thing's a setup, though. Reed goes on to note "Why would the government do this?... Because it is in the best interests of some parts of the government to have this capability. Now enough of the conspiracy theories." Nevertheless, Tor was one of the services least willing to discuss the security implications in the multifaceted nature of their service. They profess to have no insider threat, no monitoring of their network, and no form of data retention, with executive director Andrew Lewma professing to PenTest "We have no data, therefore no insider abuse." As with many things, this is a matter of definition. (*http://www.wired.com/politics/security/news/2007/09/embassy_hacks?currentPage=all*).

Tor is entirely volunteer-run, with nodes being run by private parties with an interest in protecting user privacy – and sometimes in harming it. It is accurate to say that Tor itself keeps no logs and retains no data, a benefit of its decentralized nature and lack of organizational accountability. But by devolving responsibility onto random Internet volunteers – whose reasons for volunteering, as with Reed's work on it, may stray from the interests of private users – the network resultingly has no way to guarantee the integrity of its nodes. An early beneficiary of this issue was Wikileaks (*http://decryptedmatrix.com/live/a-primer-on-wikileaks-espionage-on-tor-exit-nodes/*), which sniffed Tor exit nodes – the cleartext, final connection between

the network and the user's desired Internet endpoint. For what it's worth, Tor's Lewman has denied this connection (*http://yro.slashdot.org/story/10/06/01/2334237/*) in a public response to a 2010 article alleging same. "We hear from the Wikileaks folks that the premise behind these news articles is actually false – they didn't bootstrap Wikileaks by monitoring the Tor network. (*http://ryansholin.com/2010/05/31/wikileaks-and-tor-moral-use-of-an-amoral-system/*) But that's not the point. Users who want to be safe need to be encrypting their traffic, whether they're using Tor or not," he notes. Speaking to PenTest, he noted "Fully decentralized, anonymous networks are the only future." And there, I agree with him. Tor is not trying to be perfect today – or rather, it is, but knows that this will be a goal long in the making.

The endpoint issue is not unique to Tor by any means, but rather common to privacy providers of all stripes. VikingVPN's Derek Zimmer reinforced this point, telling PenTest "The greatest enemy of privacy for our users is at the endpoints. This includes the security of the clients, and the security of the servers at the opposite end ... no amount of "middlepoint" encryption will enable users to be secured from keyloggers installed from a bad email attachment or a government covert action that drags up all of your gmail in – even HTTPS being the default is not going to stop a determined government agency, as we have seen in their race condition exploits in the latest round of Snowden revelations."

Tor is still maturing as a service – as they themselves will tell you, the network and the software are experimental, and not meant for use in life-or-death situations. Despite this, some users are forced to do just that, and a popular dodge against network traffic analysis and exploitation is the use of Tor hidden services, colloquially known as the "deep web". (*https://www.sickchirpse.com/deep-web-guide/*) By hosting services within the Tor network, traffic avoids the cleartext endpoint problem, and the associated end-to-end encryption protects against abuse by rogue nodes. While an obscure academic Internet curiosity only a few years ago, deep web sites have come to host an endless variety of content and services that might not find so warm a welcome on the public web, perhaps most notable of the being Silk Road. (*http://www.forbes.com/sites/andygreenberg/2013/09/05/follow-the-bitcoins-how-we-got-busted-buying-drugs-on-silk-roads-black-market/*).

As one of the only privacy networks able to scale in the face of growing surveillance threats, Tor may very well be the face of things to come. Lewman adds that "Users understand their privacy needs the best. The user can decide what to disclose and to whom," and while some Tor users are aware of the shortcomings in the service (*https://blog.torproject.org/blog/plaintext-over-tor-still-plaintext*), for now it remains equal amounts privacy network and network used for analyzing the habits of users desiring privacy. Still, between recent revelations of the FBI hijacking Tor servers (*http://www.wired.com/threatlevel/2013/09/freedom-hosting-fbi/*), NSA-authored malware targeting the Tor browser bundle (*http://arstechnica.com/tech-policy/2013/08/researchers-say-tor-targeted-malware-phoned-home-to-nsa/*) and booby-trapped advertisements tracking Tor users (*http://m.cnet.com/news/nsa-tracks-google-ads-to-find-tor-users/57606178*), the onus seems to be more on the user to educate themselves about Tor's strengths and liabilities than it is on Tor to protect Internet users from themselves. As an ostensibly neutral network conduit, Tor is best used as a hop in a greater amalgamation of network privacy services – one size will never fit all.

JonDonym, a similar but more centralized service, provides limited free access and a more substantial paid anonymity service. Due to the smaller nature of its network, use of billing services, and less rigorous history of testing, it does not provide equivalent anonymity to Tor (*https://anonymous-proxy-servers.net*).

VPN SERVICES

When I started this article, I was prepared to dismiss the average commercial VPN service out of hand as a serious privacy measure. Typically offering only one hop between the user and their network endpoint, VPN's can be seen to combine some of the worst aspects of Tor into one bundle – the propensity for network analysis and operator sniffing, coupled with few of the benefits. As I began to get responses though, it became clear that VPN services have not stagnated in an increasingly adversarial network environment, but instead grown to meet some of today's threats. Like Tor, they are not a catch-all solution (*http://www.informationweek.com/security/risk-management/nsa-surveillance-can-penetrate-vpns/d/d-id/1110996?*), but rather a worthwhile step in a broader package of privacy protection. As echoed by Reuben Yap of BolehVPN, "In the era of social media and Google, user privacy is becoming more difficult to protect and often a VPN is not a complete solution."

SINGING UP

All other things being equal, the cardinal concern when signing up for a VPN service should probably be how exactly you plan to pay for it. Not so much how you'll be able to *afford* it, but the risks inherent in signing up for an anonymity service with payment methods that, particularly with post-9/11 "Know Your Customer (KYC)" rules (*http://en.wikipedia.org/wiki/Know_your_customer*), are readily tracked back to the individual user.

Credit cards

Credit cards should pretty much be a no-brainer. You get a statement in the mail, to your house, with your name on it. It's not the apex of privacy. Still, there are some modest ways to increase your financial privacy. If your privacy concerns are of the more everyday kind – ducking stalking and improving personal safety – many states have programs which allow you to keep your address confidential, even on utility bills and credit statements. One such program is California's "Safe At Home", (*http://www.sos.ca.gov/safeathome/applicants-participants.htm*) which unlike the more quotidian PO box assigns you an address indistinguishable from a normal mailing address.

This still doesn't solve the issue of your payment card being associated with your identity, however. A popular dodge for this issue involves the use of prepaid credit or "gift" cards. While re-loadable ones may perform some cursory identity checks, single-use cards are typically bought in cash and accept whatever information you elect to put on them, instead of the more onerous Know Your Customer requirements of longer-term accounts. Also, the Address Verification System used for most card transactions (*http://en.wikipedia.org/wiki/Address_Verification_System*) has the quirk of being number based. If you're John Doe at 123 Fake St, 20505, DC, USA, AVS will still approve your transaction if you say you're John NoMoe from 123 Real St in 20505, McLean, USA. You should check with both your card and your payment processor's TOS/AUP to ensure that such hijinks aren't illegal in your area, though typically such proscriptions only kick in if your use is fraudulent. While fraud will probably always be a component of anonymity services, it only helps to stigmatize them and their users, not to mention being very poor netizenship.

Bitcoin

Covering every payment processor in existence is beyond the scope of this article, but no such article would be complete without mentioning Bitcoin. Lauded by some as an anonymous, secure currency, Bitcoin has its own set of security issues of which users should be aware. (*http://bitcoin.org/en/you-need-to-know*)
For one, its anonymity only goes so far as its user's habits. While it's true that Bitcoin requires no account information other than an account code to effect a transaction, Bitcoin exchanges to need some way for you to pay them, typically require proof of identity, and each Bitcoin transaction is immortalized on public cryptographic record, known as the "blockchain" (*http://howtobuybitcoins.info*).

For the user, this means that transaction analysis is always possible – if you pay someone in exchange-bought Bitcoins, at the very least an educated guess can be made at their provenance. While Bitcoin "laundering" services exist, their antics have been known to range from stealing submitted Bitcoins wholesale to skimming their fee and returning the balance to unwary users entirely unlaundered (*http://www.wired.com/wiredenterprise/2013/08/bitocoin_anonymity/*).

At least for now, it still turns out that buying Bitcoins is best done the old fashioned way – on the down-low. Private Bitcoin traders and Bitcoin exchange mediums such as informal IRC channels thrive as ad hoc, more private ways of converting Bitcoins to cash and back. (*http://www.wired.com/wiredenterprise/2013/07/buttonwood/*) And since the nature of the Bitcoin market allows for single events or sudden trends to acutely and unexpectedly affect their value (*http://www.fool.com/investing/general/2013/10/23/bitcoin-value-bounces-back-after-silk-road-shutdow.aspx*), the currency has remained somewhat more stable than one might have expected of a system held together by user faith. (*http://blockchain.info/charts/market-price*)

Still, the onus is once again on the user to decide who to trust and what to reveal. Bitcoin exchanges have something of a history of being insecure and/or fly-by-night operations (*http://pandodaily.com/2013/11/11/fraud-at-this-chinese-bitcoin-exchange-cost-clients-4-1m-but-the-broader-market-barely-noticed/*) (*http://www.cnbc.com/id/101213462*), and individual sellers often have only the incentive of their good name to deal honestly, knowing that the individual user will have little to no recourse if they disappear with the user's coinage. And with roughly half of all Bitcoin exchanges having shut down since their inception, users may find themselves with little recourse if they want to get their Bitcoins off the market in the face of a bubble collapse. (*http://www.forbes.com/sites/kashmirhill/2013/11/15/bitcoin-companies-and-entrepreneurs-cant-get-bank-accounts/*) Bitcoins don't yet seem to be going anywhere soon, however, especially for those with anonymity concerns. As a former Wikileaks hosting provider PRQ.se tells PenTest, in the face of increasingly problematic payment processors, "[Bitcoin] is gaining more ground."

A well-informed user can still put their money on Bitcoin and have it be a good decision. As is the trend with decentralized services, it's just a matter of who to trust, how much to trust, and faith that the Bitcoin won't go the way of the Dutch tulip. (*http://www.damninteresting.com/the-dutch-tulip-bubble-of-1637/*).

What To Look For

Traditional VPN services have historically provided a modicum of encryption and a direct path off their network to whatever site the end user wants to visit. This, of course, means that a party with privileged access to the Internet backbone and ISP networks – such as a government – would have little difficulty associating traffic with a given user if they were sufficiently motivated. By combining anonymity services such as Tor with VPN's a user can enjoy more robust privacy, but what about the average user looking for a single privacy solution?

For starters, not every VPN protocol is created equal. The most common VPN protocol is PPTP, which has a history of being the most convenient and least secure to shutter your data along a private connection. Maxing out at 128 bit RC4 encryption (and sometimes implemented with as few as 40 bits), it might keep the scary hobo at Starbucks from sniffing your wireless. But don't make any long-term plans around that. (*https://www.cloudcracker.com/blog/2012/07/29/cracking-ms-chap-v2/*) Even if all goes well, should you be up against a state-sponsored adversary, your privacy expectations might not keep. NSA is known to retain some ciphertext for future decryption (*http://cryptome.org/2013/09/nsa-math-problem.htm*), and as is sometimes said by NSA cryptkeepers, "Cryptanalysis always gets better. It never gets worse."

Worse indeed, the majority of PPTP implementations have a history of failing open. This means that if routing momentarily skips a beat on your network, your traffic might go out in plaintext. Worse, if you're disconnected – and trust me, you will get disconnected – your traffic will cheerfully skip along in plaintext as well, leaving it up to you to keep an eye on the network status and hope that none of those secret recipes slip out before you can yank the Ethernet cord out of the wall. (*http://www.wilderssecurity.com/showthread.php?t=245033*) The same problem affects DNS – the Internet's equivalent of a server phone book – when encountering problems. Requests to DNS servers can consequently be leaked, allowing an adversary to get an idea of every site a user visits. While there have been incremental band-aid fixes to the protocol here and there (*https://www.schneier.com/paper-pptpv2.html*), it just can't be trusted if enough is on the line – or the wire.

Some providers have come up with homegrown solutions to these issues. At CyberGhost, CEO Robert Knapp relates that these are issues being addressed. "We added a very unique function in CyberGhost 5 that delivers secure connection and reestablishment of the connection that will not unveil anyone's anonymity during a short disconnection ... No one gets to lose his online anonymity for a second, during a disconnection from the network," adding that the DNS issue is also not lost on CyberGhost, which now funnels all VPN traffic through its own nameservers.

Similarly, LiquidVPN provides a script library for VPN clients (*http://www.prnewswire.com/news-releases/vpn-service-liquidvpn-releases-new-vpn-software-and-script-library-228982071.html*) which support script execution on connect and disconnect (such as disabling the network if traffic would go in the clear), though founder David Cox believes in an abundance of caution. "For the utmost in security we recommend using a 3rd party firewall to completely block internet access when not connected to the VPN," Cox affirms. (*http://www.liquidvpn.com/vpn-script-library/*)

L2TP is another popular protocol that provides higher cryptographic protection, but also higher CPU use. It can be considered reasonably secure under most circumstances, but is only as secure as the shared secret that protects it. The shared secret should, of course, be secret, but users should make sure that this is the case – a review of a number of VPN services found that shared secrets were sometimes provided directly on the sites, or shared between users. This vastly facilitates the decryption of traffic captured on the line, once again providing only the scary hobo level of security. Maxing out at 256 bits, L2TP provides protection against many of today's threats, but will almost certainly not prove to be future-proof.

While most VPN providers are hesitant to educate users about these shortcomings, some are more candid. LiquidVPN's Cox tells PenTest "Currently we do use L2TP but we highly recommend OpenVPN. We use shared secrets and they are rotated randomly," adding that the shared secrets are distributed through the account management area since LiquidVPN's prior encrypted e-mail method led to confused users. As long as shared secrets are not sent in the clear, L2TP is fairly robust, but most secure when employing certificate based-authentication. CyberGhost's Knapp seems conscious of these shortcomings as well, telling PenTest "We use a shared secret. It is static and listed in the users account management ... we will improve that in future."

OpenVPN is probably the most robust of the three in common use. Not all VPN services provide it, and while both PPTP and L2TP are generally well-supported on tablets and other mobile devices, OpenVPN can be a bit tricky to set up. Supporting high-end encryption and typically not beset with the traffic leakage issues of PPTP, OpenVPN supports digital certificate authentication and will keep slinging packets even on the rockiest of connections.

None of this, however, really mitigates the underlying problem of traffic analysis. But there are some good ideas. Enter once more LiquidVPN's David Cox. At LiquidVPN, Cox has introduced a technology which modulates, or rotates, the public-facing Internet addresses of users. Speaking with PenTest, Cox explained "[IP modulation] adds another layer of security over the standard shared IP address. In its simplest form it is 2 or more subnets of shared public IP addresses that randomly rotate between all the users on that VPN node making it much more difficult to track activity back to any one user."

If this sounds familiar, it should – albeit probably inadvertently. Ersatz ISP America Online employs a similar system of proxies for load balancing, a practice which caused no end of grief to users accessing systems with by-IP restrictions. (*http://en.wikipedia.org/wiki/Wikipedia:AOL*) This is not to say that Cox's idea is a bad one – quite the opposite. Rather, it speaks more to the current state of privacy affairs that an annoyance from the past has become an adroit solution for the present.

Where To Go

Different countries have varying degrees of receptivity to the idea of privacy services, ranging from outlawing to heavy surveillance to enthusiastic adoption. Services in the United States and allied countries have come under increasing suspicion from the rest of the free world following Edward Snowden's revelations this year (*http://en.wikipedia.org/wiki/Edward_Snowden*), but even before that it was known that many privacy services would bow to political and legal pressure, as Sony hacker Cody Kretsinger discovered in 2011.

Kretsinger, then 23, used the UK-based HideMyAss VPN/proxy service to launch an SQL injection attack against the electronics giant's Internet presence and was promptly shopped to the authorities by the service upon receipt of a court order, a move which some would condemn as hypocritical, but which HideMyAss at the time dismissed as their legal duty. (*http://www.theregister.co.uk/2011/09/26/hidemyass_lulzsec_controversy/*)

While HideMyAss representatives did not immediately respond to PenTest requests for comment, e-mail encryption service HushMail's Ben Cutler took a similar tack, at least by implication, telling PenTest "We keep logs for 18 months ... We never share data with third parties, other than in accordance with our privacy policy," adding "Logs are an important tool in resolving support requests, investigating and preventing abuse." Said privacy policy has previously been the basis of controversy subsequent to the company's response to subpoenas. (*http://www.wired.com/threatlevel/2007/11/encrypted-e-mai/, http://www.zdnet.com/blog/threatchaos/hushmail-betrays-trust-of-users/487*)

Not all services share HideMyAss & HushMail's attitudes towards questionable activity. While not encouraging Internet mischief, a PROXY.sh spokeswoman expressed no desire to become her users' keeper, noting "We used to have an ethical policy that would allow us to install Wireshark on a specific VPN tunnel, in order to respond to complaints related to activities directly harmful to other human beings such as paedophilia. [As of] 1 December 2013, we are no longer allowing ourselves such intervention." Similarly, BolehVPN's Reuben Yap told PenTest "At this point in time, we do not monitor any services in real time for specific types of activity," while PROXY.sh simply stated "We do not keep any logs."

Further afield, overseas VPNs are sometimes regarded as the service of choice for those concerned about cautionary tales such as Kretsigner's, but it pays to look closely before making a choice – not every legal environment is created equal. Malaysia-located BolehVPN's Reuben Yap tells PenTest that while "we do not retain data and are not required to do so via Malaysian law," the passage of Malaysia's Security Offences Special Measures Act (SOSMA) in 2012 (*http://malaysianlaw.my/attachments/Act-747-Security-Offences_85130.pdf*) "is worrying given that the Public Prosecutor requires only vague grounds to warrant interception ... we are taking steps to ensure all our servers handling customer data are located overseas so that if they require the interception, they would have to compel us to do it."

Another contender for canonical exemplar of Internet censorship, Iran deems VPNs that are not state-run illegal, and the Iranian government engaged in a brief flirtation with the idea of a nationalized VPN service that would be the only legal privacy measure, with Mehdi Akhavan-Behabadi, head of Iran's "Supreme Council of Cyberberspace" tellng the Mehr News Agency earlier this year that Iranian Internet users "have to turn to illegal VPNs to meet their needs ... so legal VPN services will be provided." However the project, which would have turned even more users towards Tor bridge nodes (*https://www.torproject.org/docs/bridges.html.en*) to escape their government's censorship, was deemed a rank failure a little over six months later with only a little over 25 companies having registered to use it.

While the United States is unlikely to overrun Iran in the Internet censorship department anytime soon, to some it's not for lack of trying. LiquidVPN founder has felt the burn, telling PenTest that "We have been mulling over the idea of moving to Sweden or Malta for some time. The reason we have not is because quite frankly being registered in Sweden or Malta means nothing unless we up and move our families there," a warning that VPN users with high-stakes Internet traffic would be wise to heed. After all, a corporate flag of convenience means nothing if the staff can be detained in contempt of court in their home nation.

Even if your VPN provider has solid security and resides in an otherwise friendly country, what do you really know about it? Free VPN services frequently publish little information about themselves (*http://www.freecanadavpn.com/*) and rarely have entirely altruistic motives. Romania is a prolific host of VPN services, (*http://www.vpnbook.com*) but also of cybercrime, with at least one Romanian town being declared something of a Mecca of Internet fraud. (*http://www.le-vpn.com/a-small-city-in-romania-is-the-world-capital-for-internet-scam/*). Neither can VPN reviews be trusted entirely – there is compelling evidence that some roundups of VPN services in the tech press may have ulterior motives, especially funneling subscribers into hand-picked services to score commissions from affiliate relationships which are not disclosed in reviews. Typically, if a site is dedicated entirely to VPN reviews (*www.vpnserviceyes.com/blog/top-5-romania-vpn/*) with no mention of authorship potential users should be wary, but even some well-known tech sites have been sucked into the cash grab at the expense of accuracy.

Still, such penny ante abuse is small potatoes compared to some of the allegations that surround larger players. Anonymizer, one of the earliest and best-known anonymity services on the 'net, quietly changed hands in 2008. The beneficiary of this acquisition was Richard "Hollis" Helms, former chief of the CIA's National Resources Division (*http://www.zdnet.com/examining-the-ties-between-trapwire-abraxas-and-anonymizer-7000002770/*), and presumably proud owner of a complex nesting of companies best known for their association with the controversial Trapwire system. (*http://www.networkworld.com/community/blog/anonymizer-tied-company-selling-trapwire-surveillance-governments*) Given the complex chain of corporate ownership and the common controlling interest, the best thing that can be said about the arrangement is that Helms would probably have been happier if it had never been identified at all.

If all this seems daunting – trust me, I know. When I first started writing this piece, I envisioned it as a comprehensive review of services and software that users would be able to take into account when selecting an anonymity service. It still could have been, but it would have been an overwhelmingly dishonest one. Even in 4000 words, enough cannot be said about the complexity of Internet privacy to do it justice, and trying to come

to some sort of definitive conclusion about best practices would inevitably only serve a specific set of people. There's no single, good solution. The point of all this isn't to scare users away from anonymity services. If I haven't given them a rose-tinted evaluation, that's probably because I've got nothing to sell.

The world of privacy as a service is a complex, sometimes even scary landscape of services which are often poorly defined and often benefit from an even greater lack of accountability than the users they protect. By no means is this true of all, but there's a lot more to think about than price and convenience when settling on one to use. I make no recommendation because there's no single best product to recommend.

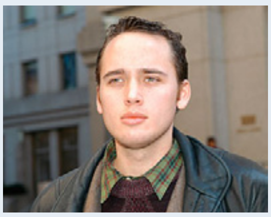
The Internet and its users, who have long enjoyed having their privacy and security be someone else's problem, need to start taking responsibility for their own security. Is your GMail insecure? Don't wait for Google to fix it. ISP blocking your torrents? Don't wait for legislation to make it better. Such things are inevitably in a constant game of catch-up against the last threat, just in time to be bowled over by the new one. Legislation has never solved a problem on the Internet. For that matter, when has relying on a corporation to take your best interests to heart?

Each user has a responsibility to maintain their own virtual situational awareness, their own threat model, and their own plans to ensure their best interests. No one else is going to do it for them, and the one security measure I really *can* endorse is "Stop waiting for someone else to."

Suggested extensions

- NoScript: This extension gives you granular control over which scripts you allow to run in your browser, allowing you to avoid privacy-impairing tracking measures. *https://addons.mozilla.org/en-US/firefox/addon/noscript/*
- AdBlock: Similarly, AdBlock allows you to protect yourself against the malicious use of advertising networks. *https://adblockplus.org/*
- Geolocater: This extension gives you room to breathe by dodging some types of geolocation services. *https://addons.mozilla.org/en-US/firefox/addon/geolocater/*
- FoxyProxy: This tool is handy for managing various forms of proxy servers, letting you protect yourself against inadvertent IP address disclosure. Never use public or open proxy servers for privacy purposes – you don't know who runs them or why, and they almost always log. *http://getfoxyproxy.org/mozilla/standard/install.html*
- Web Developer: With this extension you can perform easy and in-depth analysis of what's going on behind the scenes on web pages. *https://addons.mozilla.org/en-US/firefox/addon/web-developer/developers*
- Disconnect: Another way of dodging tracking networks. *https://disconnect.me/*
- CryptoCat: Browser-based encrypted chat. *https://crypto.cat/*
- HTTPS Everywhere: Whenever possible, keep your connections to web sites encrypted. *https://www.eff.org/https-everywhere*

About the Author



Adrian Lamo is a threat analyst, writer, and former computer hacker with 15 years' experience in hacker issues, practices, and culture. Between having spent a time in fugue from the FBI in 2003 and playing a key role working with the military mitigating the largest national security breach in US history in 2010, the only true constant in his life is strange luck. He lives in the greater Washington, DC area, and can be found on Twitter at https://twitter.com/6.


KOENIG
WORLD'S #1 OFFSHORE TRAINING
Be Certified
& Expert in
Web Apps
Security
Linux
info@koenig-solutions.com
www.koenig-solutions.com
+91 88009 40444